



# Introduction to Embedded Linux

---

## *Student Guide*



*Revision 3.1  
October 2014  
February 2015*



## Important Notice

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Copyright © 2012 Texas Instruments Incorporated

## Revision History

August 2012    – Revision 1.0  
April 2013     -- Revision 2.0  
October 2014   -- Revision 3.0  
February 2015 -- Revision 3.1

## Mailing Address

Texas Instruments  
Training Technical Organization  
6550 Chase Oaks Blvd  
Building 2  
Plano, TX 75023

# Module 01: Booting Linux

---

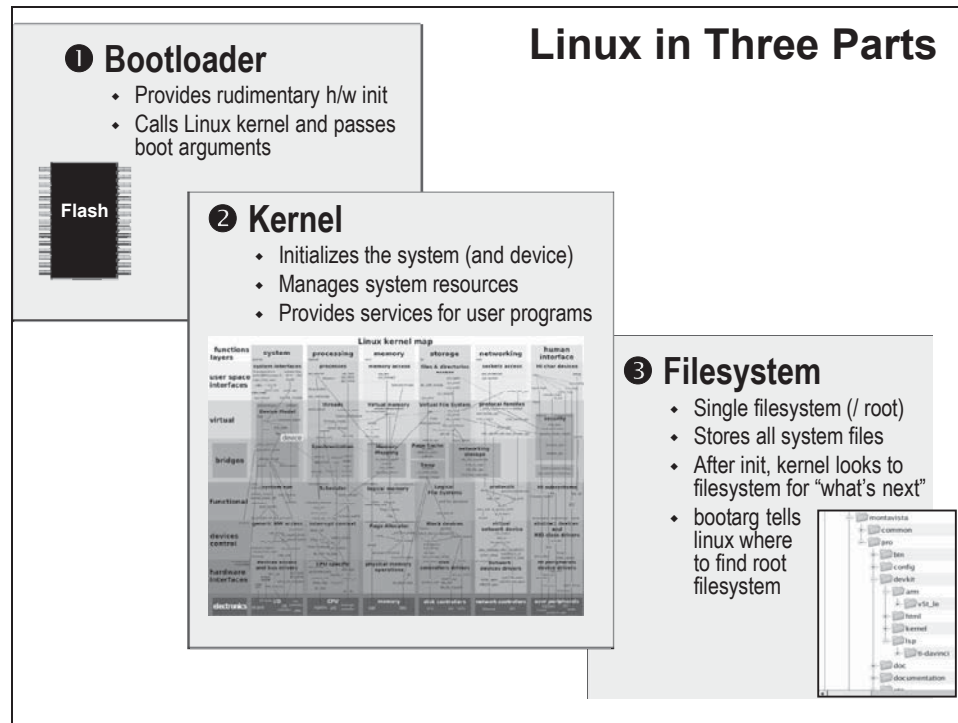
## Introduction

This module begins by showing how a Linux distribution may be booted using a provided u-boot bootloader, Linux kernel and Linux filesystem. Next it shows how those same three elements may be rebuilt from source code and possibly reconfigured. Finally, the module covers some details of the “System five” (sys-V) initialization standard that is used to specify the startup configuration of many Linux distributions, including Arago (Beaglebone distribution) and Ubuntu (x86 Distribution.)

## Module Topics

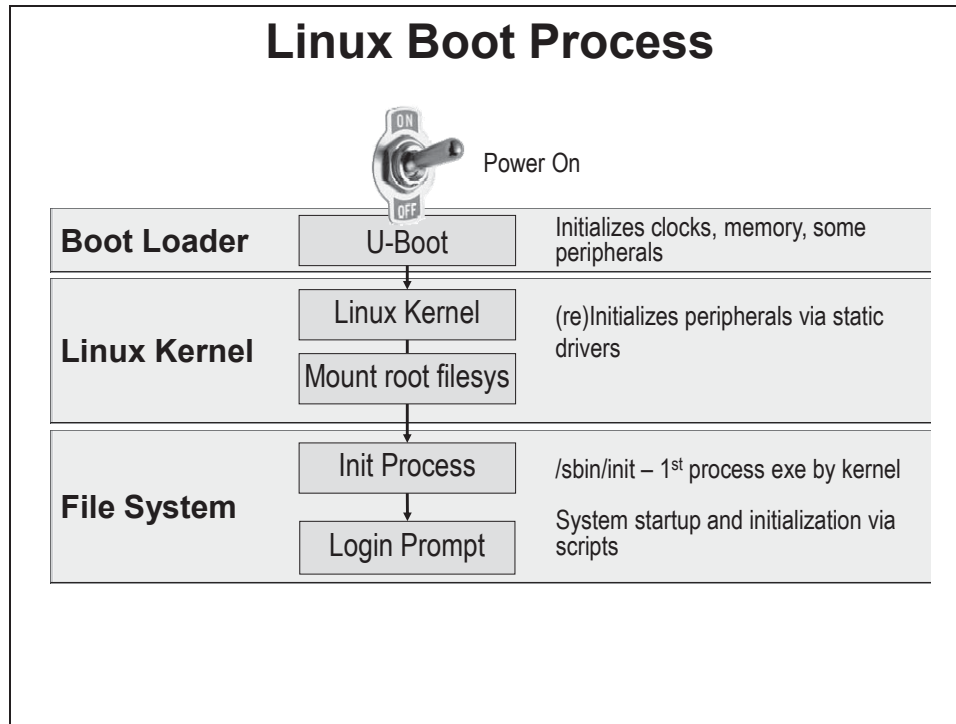
|  |             |
|--|-------------|
| <b>Module 01: Booting Linux .....</b>                | <b>1-1</b>  |
| <i>Module Topics.....</i>                            | <i>1-2</i>  |
| <i>Booting Linux from Pre-Built Binaries.....</i>    | <i>1-3</i>  |
| <i>Rebuilding and Modifying SDK Components .....</i> | <i>1-12</i> |
| <i>System Startup Details .....</i>                  | <i>1-19</i> |
| <i>Lab 1 Introduction .....</i>                      | <i>1-26</i> |

# Booting Linux from Pre-Built Binaries



The Linux operating system contains components in both the Linux kernel and the root filesystem. These components are cross-dependent. For instance, Linux drivers contain program code that exists in the kernel as well as device nodes that exist in the “/dev” directory of the root filesystem. As such, a Linux system cannot complete its boot procedure without both the kernel and root filesystem present.

The Linux kernel has the ability to boot itself from RAM memory (such as DDR), but does not contain code to re-locate itself from non-volatile memory into RAM. Thus, a standalone boot loader is also required. A number of bootloaders have been developed in the Linux community for this purpose, including u-boot, grub and LILO. Embedded systems typically use u-boot, which is the smallest and simplest of the three listed. Grub and LILO are more commonly employed in desktop systems.



The first code to execute at startup is the boot loader. As will be shown in the following slides, the software bootloader, i.e. u-boot, is actually loaded by a ROM bootloader that is factory programmed onto the device. U-boot performs some low-level initialization, but generally will only initialize hardware as required to perform its duties. For instance, if u-boot is to copy the Linux kernel from NAND flash into DDR, it will initialize the Asynchronous memory interface and the DDR interface, as well as performing some low-level initialization, such as configuring the clock tree and the watchdog timer.

Once the bootloader has copied the kernel to DDR, it calls a startup routine called “setup()” which has code to decompress the compressed kernel image and launch the “start\_kernel()” routine to begin booting. As the kernel boots it will initialize the hardware corresponding to any driver that has been statically built into the kernel image. In some cases, this will be a re-initialization of hardware that was previously configured by u-boot.

After the kernel has completed its boot procedure, it calls a special application (also known as a Linux “process”) from the root filesystem. By default this application is located in the subdirectory “/sbin” (for “system binaries”) and is named “init.” This initialization process completes the boot procedure and performs final system initialization, usually via configuration files and scripts located in the root filesystem. Different methodologies may be employed for this configuration. The most common, which is used both in Ubuntu and TI’s Arago distribution, is named “System 5.” (abbreviated “sysV”) System 5 will be discussed at the end of this module.

Generally speaking, all hardware-specific initialization should be performed by the Linux kernel via drivers, and the remaining, non-hardware-specific initialization is performed in the filesystem. Ideally this isolates the effort of porting of a Linux distribution across hardware platforms to modifications only within the Linux kernel, allowing the entire root filesystem to be ported unchanged, except for a re-compilation of any binary files.

## Where Do You Find ...

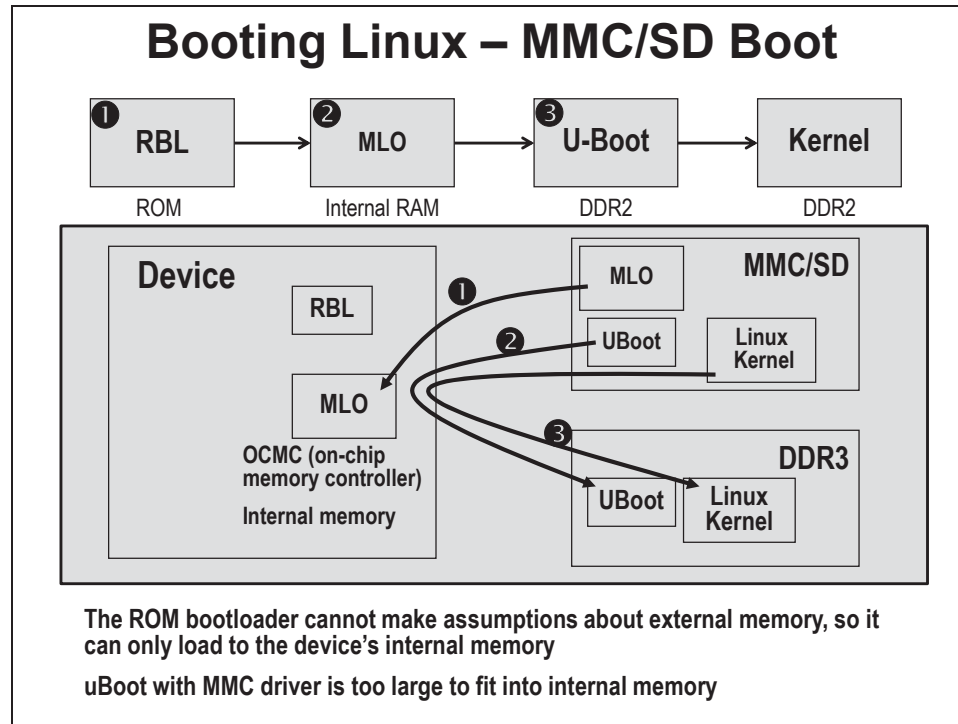
| Where located:              | Flash Boot   | MMC Boot   |  |
|-----------------------------|--------------|------------|--|
| 1a. <b>SBL</b> , UBL or MLO | n/a          | <b>MMC</b> |  |
| 1b. Bootloader (U-Boot)     | <b>Flash</b> | <b>MMC</b> |  |
| 2. Linux Kernel             | <b>Flash</b> | <b>MMC</b> |  |
| 3. Filesystem               | <b>Flash</b> | <b>MMC</b> |  |

- ◆ Multimedia Card (MMC) or Secure Digital card (SD) may be flashed using an MMC/SD programmer using a variety of utilities
- ◆ Simple, low cost method for booting Linux (or just U-boot) on a development board that has nothing pre-programmed (or for recovery.)

The Linux operating system operates from RAM memory, typically DDR (Double Data-Rate) SDRAM. In order to boot the system, the bootloader, kernel and filesystem must be stored in non-volatile memory. Many different configurations are possible. This slide shows two of the most commonly used storage configurations – NAND flash and MMC/SD card.

When booting from NAND flash, the bootloader, kernel and filesystem are simply written to the NAND at pre-determined offsets. By contrast, Secure Digital (SD) and Multimedia Card (MMC) interfaces use a filesystem to manage their contents. The ROM bootloader of the AM335x can parse the FAT32 filesystem, so this must be used.

Note the additional software component required for the MMC/SD boot, which is listed as “1a,” and may be referred to in TI’s documentation as either the SBL (Secondary Boot Loader), UBL (User Boot Loader) or MLO (Mmc LOader file.) As will be detailed on the next slide, this additional software component is required when the bootlader (i.e. u-boot) is too large to fit in the internal memory of the AM335x device.



This slide shows further memory details of the boot process. There are actually three bootloaders used in this example: the RBL (ROM bootloader), MLO and u-boot. As the name implies, the RBL is programmed at the factory during the manufacture of the device. In fact, it is implemented as semiconductor gates, and is represented in the mask set that is used in the photolithography mass production of the devices.

The purpose of the RBL is to load and execute a small program from non-volatile memory. The RBL of the AM335x family of devices will only load a program into the internal memory of the device, specifically the L3 RAM, which is 64K on these devices. While this is not a hard requirement for any ROM bootloader, it was the design choice implemented in this bootloader. As such, if u-boot is larger than 64K in size, the RBM must first load a secondary bootloader into internal memory (step 1 as shown) and the secondary bootloader then initializes the DDR interface and loads the u-boot program into the DDR memory (step 2.) Finally, u-boot loads the Linux kernel into DDR and calls the “setup()” function from the kernel to pass control to the kernel which will then decompress itself and begin executing.



## Where Do You Find ...

| Where located:              | Flash Boot   | MMC Boot   | NFS Boot     |
|-----------------------------|--------------|------------|--------------|
| 1a. <b>SBL</b> , UBL or MLO | n/a          | <b>MMC</b> | bootp / tftp |
| 1b. Bootloader (U-Boot)     | <b>Flash</b> | <b>MMC</b> | tftp         |
| 2. Linux Kernel             | <b>Flash</b> | <b>MMC</b> | tftp         |
| 3. Filesystem               | <b>Flash</b> | <b>MMC</b> | nfs          |

- ◆ NFS boot is typically used for development but not production devices
- ◆ All components of the system are loaded from host server at each boot
- ◆ Filesystem changes on host are instantly reflected
- ◆ UBL, U-boot and Linux Kernel changes are reflected on each reboot
- ◆ Good method to ensure uniformity across multiple development boards

Another configuration is to boot Linux completely from the Ethernet connection. The RBL supports booting from Ethernet using the bootp (boot protocol) and tftp (trivial file transfer) protocols. The u-boot and kernel are also loaded using tftp, and finally the root filesystem is mounted using a Linux-style network file share protocol called nfs (network file share.)

This boot method is rarely useful for released-to-market products, but can be very useful for debugging, particularly during the board porting effort. The Ethernet connection is generally one of the simpler interfaces to bring up during a board port because the media access controller (MAC) is integrated onto the device and the MII interface between the MAC and PHY is fairly standardized even if a different PHY layer is chosen than that which is provided on TI reference designs. Once the Ethernet connection is validated, an entire Linux operating system can be loaded onto the custom board, providing much greater visibility into the system, which is of great utility in porting and validating the remainder of the system.

## Step 1: Setup EZSDK

### Setup and Rebuild Linux EZSDK

```
host $ sudo apt-get update
host $ cd ti-sdk-am335x-evm-xx.xx.xx.xx
host $ sudo ./setup.sh
```

**Installs tftp server, nfs server , minicom and packages needed to rebuild u-boot and kernel (using Aptitude package manager)**

**Extracts AM335x Linux filesystem to specified location (default \${HOME}/targetfs), exports it via nfs server, and updates Rules.mak EXEC\_DIR variable**

**Configures tftp server with specified read directory (default /tftpboot) and copies in prebuilt Linux kernel**

**Configures minicom for TTYS0 serial device for 115,200 Baud, 8-bit, no parity, 1 stop bit**

**Creates setup.minicom script to configure u-boot for desired boot type (mmc or tftp/nfs).**

The first step in any Linux development on a Texas Instruments device is to install the Software Development Kit. After de-archiving the SDK, the “setup.sh” shell script should be executed in order to complete the installation. Shell scripts are text files containing instructions in the form of commands and variables that are executed by the Linux shell.

The “setup.sh” shell script installs and configures a number of packages in the Ubuntu Linux system, including tftp and nfs servers for network-based booting, and the minicom utility for communicating with the target board over UART. These packages are installed using the “aptitude” package manager, so the “apt-get update” command needs to be executed before running the script. Because this command requires root permissions, the “sudo” statement (“switch user do” or “super user do” depending on who you ask) should be prepended to the update command.

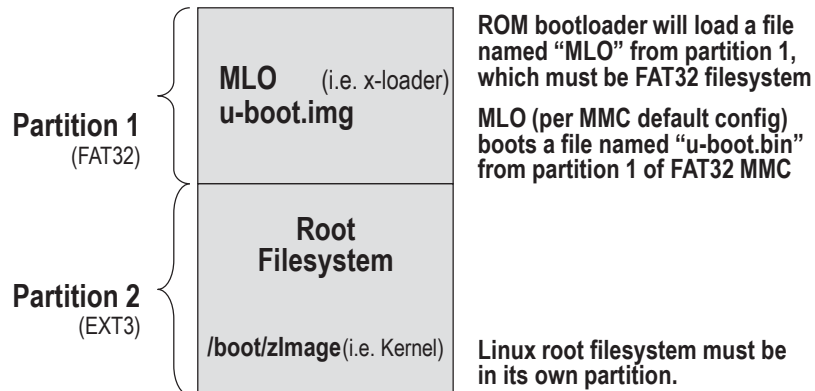
## Step 2: Boot Linux from MMC

### MMC Boot

Attach SD/MMC programmer with inserted Micro-SD adapter

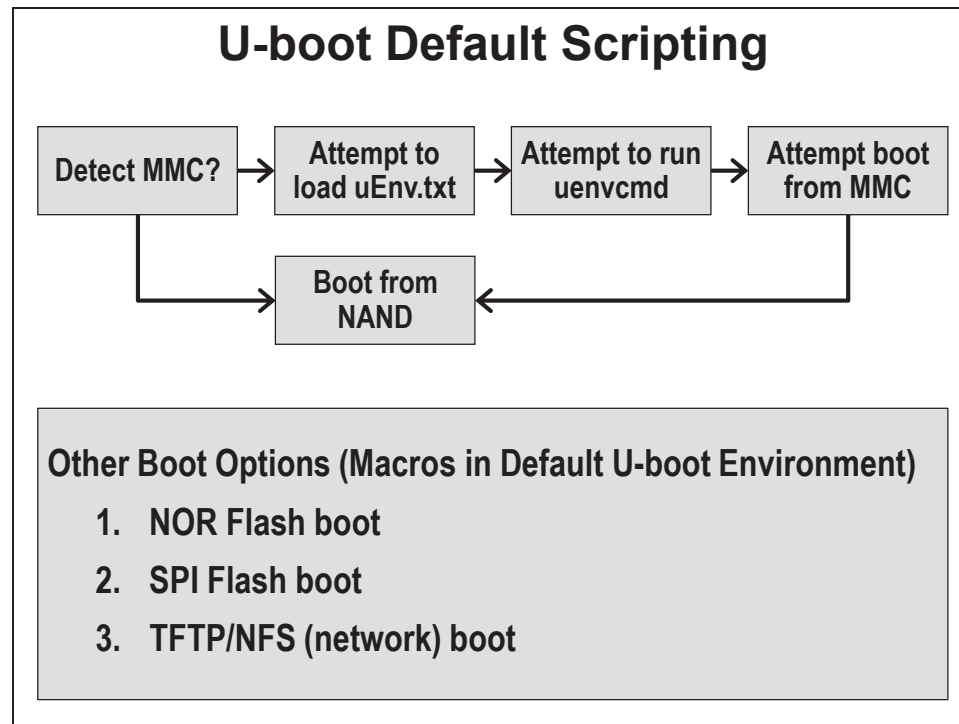
```
host $ cd ti-sdk-xx.xx.xx.xx/bin
```

```
host $ sudo ./create-sdcard.sh
```



The SDK also contains a number of useful utilities in the “bin” (binaries) subdirectory. One of these is the “create-sdcard.sh” script. This script will format a secure digital (SD) or multimedia card (MMC) as a bootable Linux disk, installing either the base SDK Linux image or a Linux image specified by the user. The utility creates two partitions on the SD card. The first partition contains the MLO and u-boot files. The second partition contains the root filesystem, as well as the Linux kernel, which is stored in the “boot” subdirectory of the root filesystem.

The first partition must be formatted with the “fat32” filesystem, because this is the only filesystem that the ROM bootloader is capable of parsing. The second partition can be formatted with any filesystem that both u-boot and the Linux kernel are capable of parsing. A common embedded Linux filesystem for SD cards and hard disk drives is the “ext3” filesystem.



The behavior of the u-boot bootloader is controlled using environment variables. The default values of the environment variables are set via a configuration file at compile time, and can be overridden by “setenv” and “saveenv” commands in the u-boot command mode.

The above flow chart shows the behavior of u-boot’s default scripting as set in the Texas Instruments SDK. u-boot will first attempt to boot from the SD/MMC interface and if this fails will attempt to boot from NAND flash. If booting from the SD/MMC interface, u-boot will search for a file named “uEnv.txt” which can be used to override the default environment variables. When booting from NAND, the override variables are saved at a set memory location.

Additional boot methods are also supported. The default environment has macros that can be used to boot from NOR flash, SPI flash and over Ethernet.

## Linux Boot Arguments for MMC Boot

```
mmc_load_uimage=
    fatload mmc 0 0x80007fc0 uImage
```

uBoot will attempt to load Linux kernel binary with filename “uImage” from mmc/sd partition 1, which must be fat32 formatted.

Kernel image will be loaded into physical address 0x80007fc0, an offset into DDR3. Subsequent call of “bootm” will boot the kernel from DDR3.

```
bootargs=
    console=ttyO0,115200n8 bootdelay=3
    root=/dev/mmcblk0p2 rw rootfstype=ext3 rootwait
    ip=None
```

Linux boot arguments set up a serial console for login and specify the root filesystem location as mmc/sd device 0 partition 2, which must be ext3 formatted.

“ip=None” allows boot scripts in filesystem to define ip settings (/etc/network/interfaces).

bootdelay and rootwait provide delays for mmc.

The u-boot environment can store variables and macros, where macros are just variables that hold instructions that can be executed using the “run” command. The first example shown here, “mmc\_load\_uimage” is a macro that loads a Linux kernel from a SD or MMC device. The macro uses the “fatload” command, which loads a file from a fat32 filesystem. The first parameter indicates to load from the mmc device, the second parameter that this is mmc device 0, the third parameter is the physical memory address to load into and the fourth parameter is the name of the file to load.

The second example shows the “bootargs” environment variable. Certain variable names are recognized by u-boot and used in u-boot function calls. This is one of those variables. When the “bootm” (boot from memory) command is used to boot the Linux kernel, the “bootargs” variable is automatically passed to the kernel as a string value. The various assignments are parsed by the kernel as it boots and modify the boot behavior. For instance, the “root=/dev/mmcblk0p2” setting tells the kernel that the root filesystem is located in the mmc device 0, partition 2.

## Rebuilding and Modifying SDK Components

### Rebuilding EZSDK

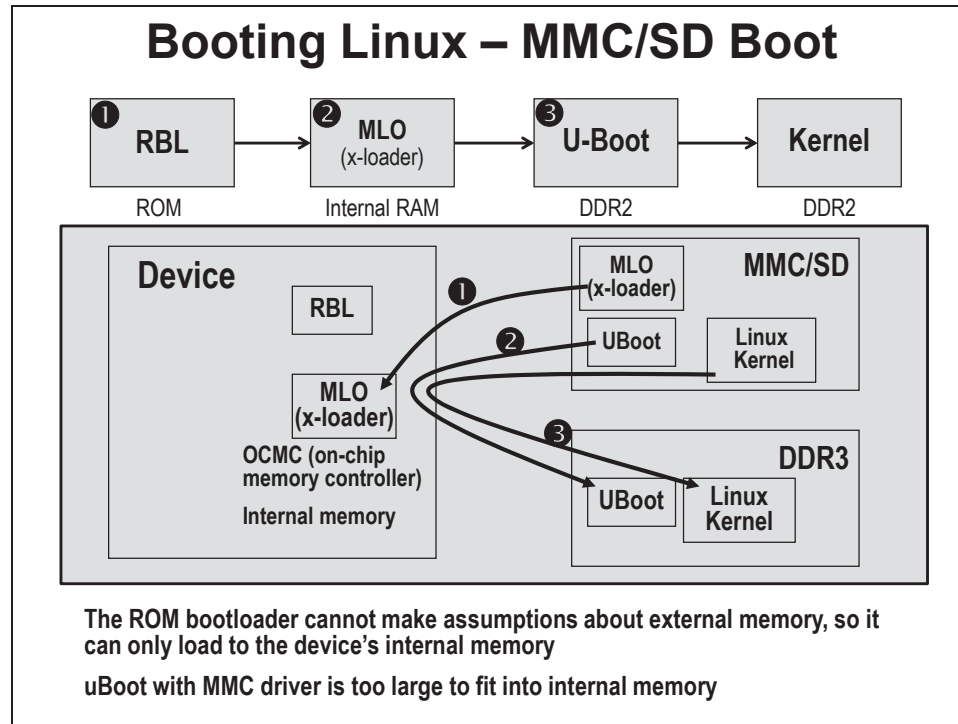
**Rebuild SDK**  
`host $ make help`  
`host $ make clean`  
`host $ make all`  
**Install rebuilt components to export directory (default \${HOME}/targetfs)**  
`host $ make install`

**make all rebuilds:**  

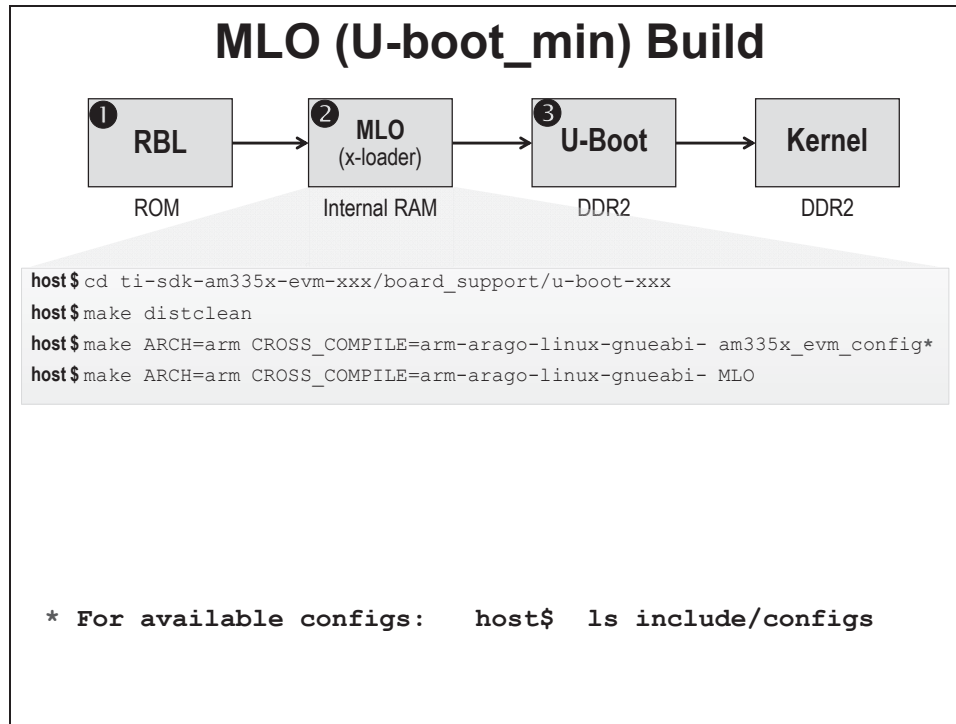
|                              |                                |
|------------------------------|--------------------------------|
| <b>U-boot</b>                | <b>Example Applications</b>    |
| <b>Linux Kernel</b>          | <b>Wireless Driver Support</b> |
| <b>OpenGL graphics Demos</b> |                                |

The simplest method for rebuilding the bootloader and Linux kernel is to rebuild the entire software development kit. This can be done from the top-level makefile using the command “make all.” Additionally, the individual components of the SDK can be rebuilt separately. The commands for each component are specified in the help listing, which can be accessed using “make help.”

Rebuilding the SDK from source is a good “reality check,” but has the disadvantage that it does not give the user the opportunity to reconfigure any of the components being built. The following slides show how each component can be built from the included source code, which gives the user the opportunity to reconfigure each component.



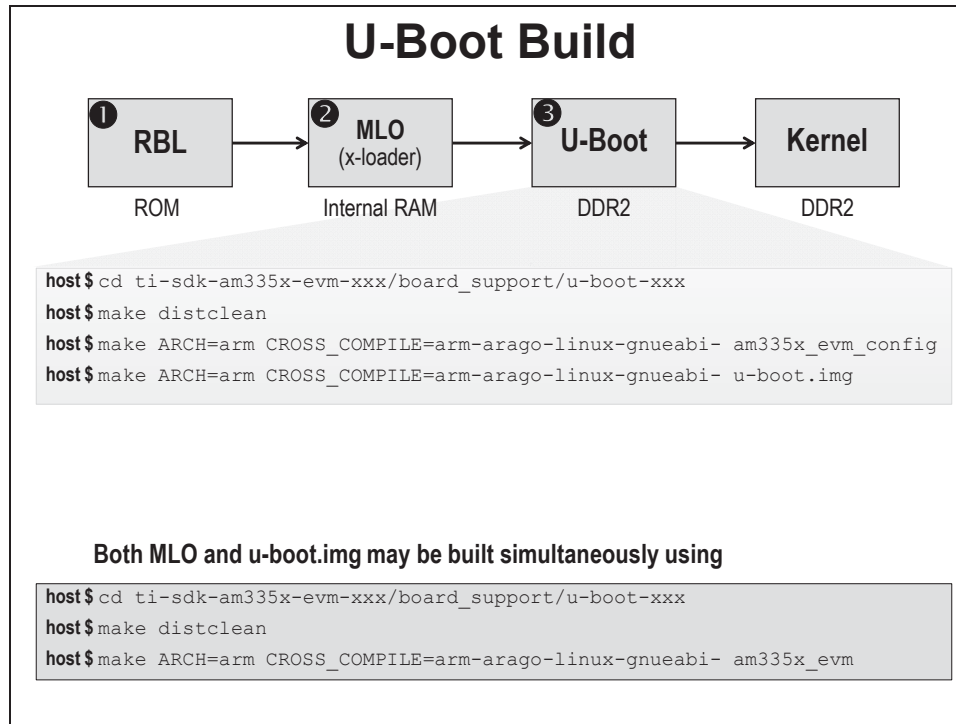
Recall that the procedure for booting the kernel from the SD/MMC interface requires an MLO file, a u-boot binary and a kernel image.



The first of these components, the MLO (Mmc LOader file), is actually built from the u-boot source code. The MLO is really just a u-boot stub. The steps to rebuild the MLO are as shown. First, change into the u-boot source code directory. Second, remove any previously built files and configuration using the “make distclean” (make distribution clean) command. The code is then configured for the appropriate target by loading a default configuration, in this case “am335x\_evm\_config.” The default configurations are stored in the “include/configs” subdirectory. Finally, the MLO binary is built using “make MLO.”

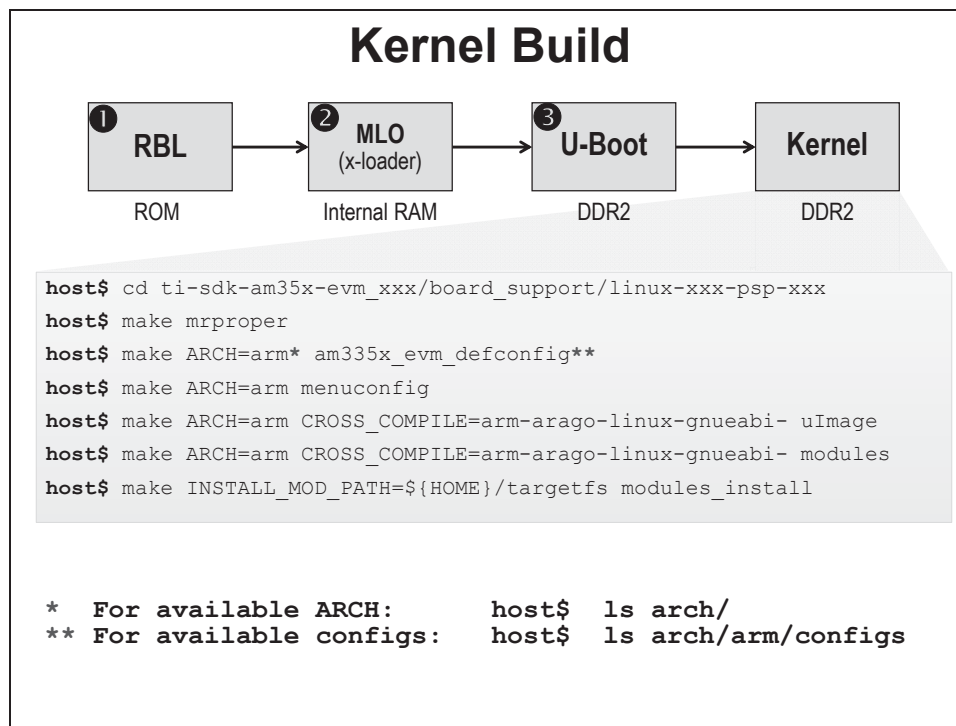
Note the “ARCH” and “CROSS\_COMPILE” environment variables. The “ARCH” variable specifies the architecture that the MLO is built for, in this case “arm.” If nothing is specified, the makefile assumes the architecture is the same as that of the host computer, which is generally an “x86” architecture. The “CROSS\_COMPILE” variable specifies the prefix of the gcc cross-compiler to be used to build the binary. A prefix of “arm-arago-linux-gnueabi-” indicates that the compiler chain to be used is named “arm-arago-linux-gnueabi-gcc”





U-boot is built almost identically to the MLO, except that the final command is “make u-boot.img”

The MLO and u-boot can be built together using a simplified command, “make am335x\_evm” This command will load the “am335x\_evm\_config” default configuration and then build both the MLO and u-boot binary.

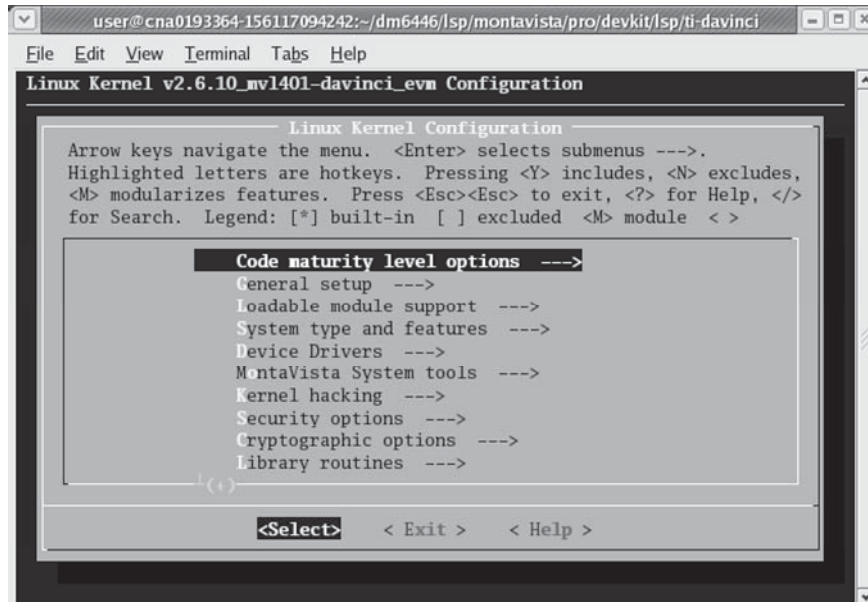


The build procedure for the Linux kernel is similar to that of u-boot although it has a few extra steps. As with u-boot, the first step is to change into the source directory and perform the clean step, which, for the Linux kernel, is called “make mrproper”

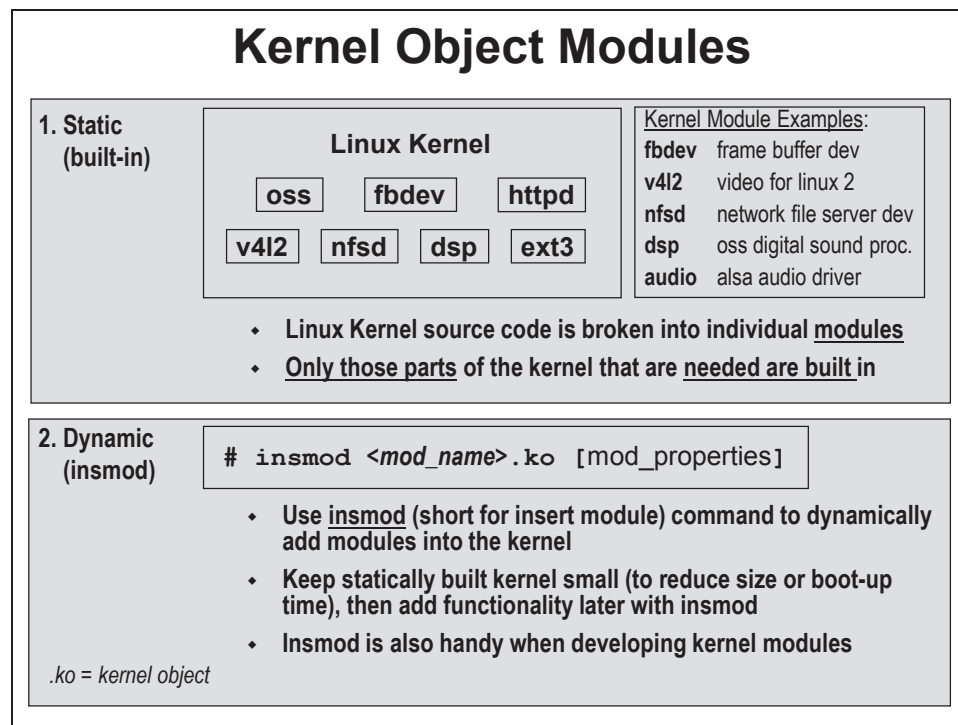
Next the default configuration is loaded, in this case “am335x\_evm\_defconfig” The Linux kernel also has an additional, optional step of “make menuconfig” which allows the user to modify the default configuration using a menu utility, as shown on the next slide. After the desired configuration is set, the kernel image can be built using “make uImage” the “uImage” or u-boot image is a compressed kernel with an additional u-boot header. Another common build step would be “make zImage” which builds the compressed image without the additional header.

After building the kernel image, the “make modules” command builds the dynamically loadable kernel modules which were configured during the “make menuconfig” step. Finally, the “make modules\_install” command copies the dynamically loadable modules to their proper location in the root filesystem.

## “make menuconfig” options

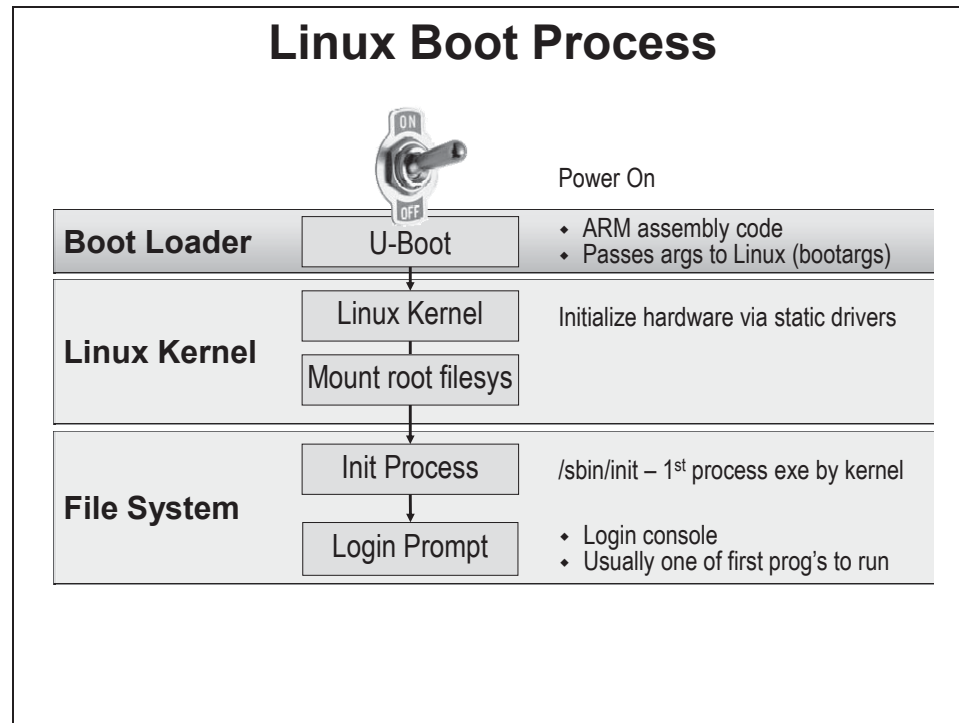


This slide shows the screen presented by the “make menuconfig” command. This is the high-level menu showing the categories of kernel modules available. After selecting category and sub-category, one descends to a list of kernel modules. As shown in the top menu, one can enter “Y” to statically build the module into the kernel, “N” to completely exclude the module from the build, or “M” to build a dynamically loadable module.



Dynamically loadable kernel modules may be loaded at runtime using either the “insmod” or “modprobe” commands.

## System Startup Details



This diagram shows the basic Linux boot flow from power-on to login prompt. The bootloader is the first block of code to be executed. As will be shown and explained momentarily, there may be as many as three bootloaders (one ROM bootloader and two software bootloaders) in the system.

The bootloader performs only that system initialization that is required for it to complete its task of loading the Linux kernel into DDR memory. This includes low-level initialization such as setting up the clock tree and watchdog timer, as well as configuring those peripherals corresponding to locations used to store the kernel and initialization of the DDR interface.

As the kernel boots, one of its primary responsibilities is to initialize system hardware. Any hardware associated with a driver that has been statically built into the kernel will be initialized at boot time. (The hardware that is associated with dynamically loadable driver modules is not initialized until the driver module is loaded using “insmod” or “modprobe.”)

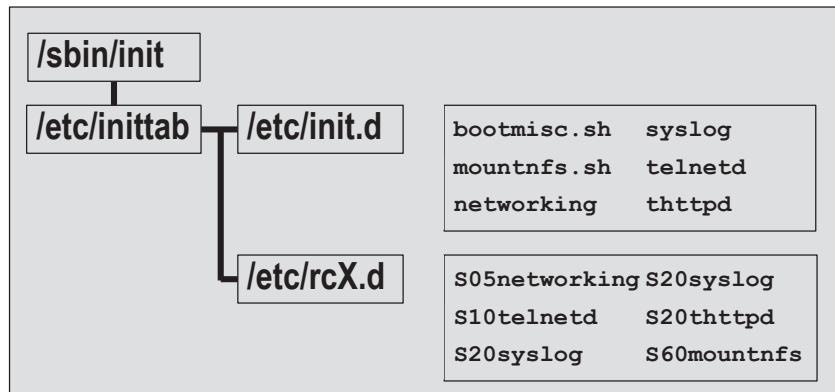
After the kernel has initialized all peripherals, it may mount the root filesystem. Naturally, the driver that is associated with the peripheral on which the root filesystem is stored must be statically built into the kernel in order for this operation to succeed. The final system initialization is driven out of the filesystem by the initialization process, by default an application that is stored in the root filesystem at “/sbin/init”

The initialization handled by the Linux kernel drivers is generally hardware specific, whereas the initialization specified in the root filesystem is generally not. This allows, at least ideally, for a Linux distribution to be ported between different hardware platforms by modifying only the Linux kernel and retaining the root filesystem unchanged. (Of course any binary files in the root filesystem would need to be recompiled if the processor core has changed.)

## Linux Initialization Process

The initialization process is launched by the Linux Kernel after the root filesystem is mounted

By default it is located at `/sbin/init`, but it can be specified as a kernel boot argument



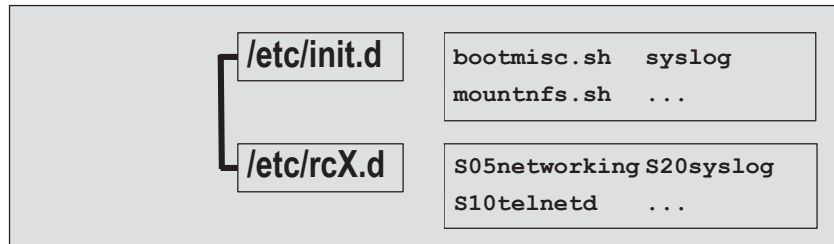
After the root filesystem has been mounted, the Linux kernel launches the initialization process. This process is the first process (i.e. program) executed by the system, and all other processes are forked from this initial process. (More on forking will be covered in module 07, which covers the Linux scheduler.)

The initialization process can execute any program that the distribution developer chooses. The most common initialization process in current Linux distributions follows the “system five” (abbreviated sysV) standard. Both Ubuntu, which is the desktop distribution used in this workshop and Arago, which is the embedded target distribution used, follow sysV.

The sysV initialization protocol begins with the “inittab” file in the “etc” directory. This is a text file that defines a number of runlevels, typically 5. Each runlevel specifies a functionality set. For instance, many systems make runlevel 1 a “safe boot” runlevel that boots the system with the minimal features – no networking and no graphics with a text login. Runlevel 5 is typically a full system boot with all of the supported features, and the runlevels inbetween are intermediate functionality sets.

The default runlevel is set in the “inittab” file. Usually this is runlevel 5, the full set of supported features. The runlevel can be overridden at boot time by the boot arguments that are passed from the bootloader to the Linux kernel.

## SysV Startup



- ◆ Arago and most Linux distributions use the SysV (“System Five”) startup process
- ◆ `/etc/init.d` directory holds all scripts used for boot-up
- ◆ `/etc/rcX.d` (i.e. `/etc/rc5.d`) contains links to these scripts specifying which should run and in what order for a given run level 1-5 (5 is default)
- ◆ `Sxx` precedes the name of each link in `rcX.d`. “S” indicates it is run at startup, “xx” indicates the order

The implementation of the runlevels is accomplished by a set of scripts in each runlevel directory. The runlevel directories are located at `/etc/rcX.d` where the “X” is the runlevel number, i.e. `/etc/rc5.d`

Whichever runlevel is selected (the default runlevel, unless overridden by the bootloader), the scripts contained in the corresponding directory are executed. Note that if runlevel 5 is selected, only the scripts in the “rc5.d” directory are executed, as opposed to a hierarchical system where the scripts in “rc1.d” through “rc5.d” are executed in sequence. (A special directory `rcS.d` is also executed regardless of runlevel, as discussed on the next slide)

Generally this means that the scripts in “rc1.d” are also in “rc2.d” – “rc5.d”. Since it would be inefficient to repeat the same script multiple times, and would also make it more difficult to update a script as all copies would have to be updated, soft links are typically used. With soft links, all scripts used for startup are placed into the `/etc/init.d` directory. Each of the “rcX.d” directories then contains soft links back to the subset of those scripts that are used for that particular runlevel. This way there is only one copy of each script, but any combination can be specified for a given runlevel.

Note that in order for a script in one of the runlevel directories to be run at startup, it must start with a capital letter “S” and two numbers. The “S” indicates that it is a startup script, and the two digit number specifies the order that the scripts are run in. It is possible for two scripts to have the same two digit number, which indicates that the order of execution of scripts at that same number does not matter.

## Beaglebone Arago Runlevel 5 Startup

### /etc/rcS.d

|              |                |                         |                |
|--------------|----------------|-------------------------|----------------|
| S01psplash   | S10checkroot   | S37populate-volatile.sh | S45mountnfs.sh |
| S02banner    | S12udev-cache  | S38devpts.sh            | S55bootmisc.sh |
| S03sysfs     | S20modutils.sh | S39alsa-state           | S99finish.sh   |
| S03udev      | S30ramdisk     | S40configure            |                |
| S06alignment | S35mountall.sh | S41networking           |                |

### /etc/rc5.d

|             |             |                   |                        |
|-------------|-------------|-------------------|------------------------|
| S10dropbear | S20syslog   | S70lighttpd       | S98storage-gadget-init |
| S10telnetd  | S20thttpd   | S97matrix-gui-2.0 | S99gplv3-notice        |
| S20netperf  | S30pvr-init | S98parse-ip       | S99rmnologin           |

**Sxx scripts are executed when entering a runlevel, according to the order of “xx” (01 first)**

**Kxx scripts are executed when exiting a runlevel, according to reverse order of “xx” (99 first)**

In addition to the runlevel directories, there is a special system directory at “/etc/rcS.d” on many systems, including the Arago distribution. All scripts in the “rcS.d” directory are executed regardless of the runlevel booted into, and all scripts in this directory is executed before any script is executed from the runlevel directory.

The slide above shows the sequence of scripts that would be executed on the Arago distribution for runlevel 5. First, all scripts in the “/etc/rcS.d” directory are executed, beginning with “S01psplash” and ending with “S99finish.sh” and then all scripts in the “/etc/rc5.d” directory are executed, beginning with “S10dropbear” and ending with “S99rmnologin”



## /etc/inittab

```
# The default runlevel.
id:5:initdefault:
# Boot-time system configuration/initialization script.
# This is run first except when booting in emergency (-b) mode.
si::sysinit:/etc/init.d/rcS
# Run-level configuration/initialization scripts
l0:0:wait:/etc/init.d/rc 0
l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
# Other Initializations
S:2345:respawn:/sbin/getty 115200 ttyO0
```

**Format:**  
<id>:<runlevels>:<action>:<process>

This slide shows the “/etc/inittab” used in the Arago distribution. The lines that begin with the hash (#) are comments.

The first programmatic line declares runlevel 5 as the default.

The next line declares the system directory “rcS” and the next six declare the startup directories for runlevels 0 through 5. (Runlevel 0 is used for shutdown on this system.)

The final line configures a login terminal for runlevels 2-5. (The “getty” program is a serial login terminal program) and configures it to use “/dev/ttyO0” device node, which is the serial port, at 115200 baud.

## **/etc/fstab**

Format:

<device>:<mount point>:<filesystem type>:<options>:<backup>:<check>

**# stock fstab - you probably want to override this with a machine specific one**

|        |               |        |                   |     |
|--------|---------------|--------|-------------------|-----|
| rootfs | /             | auto   | defaults          | 1 1 |
| proc   | /proc         | proc   | defaults          | 0 0 |
| devpts | /dev/pts      | devpts | mode=0620,gid=5   | 0 0 |
| usbfs  | /proc/bus/usb | usbfs  | defaults          | 0 0 |
| tmpfs  | /var/volatile | tmpfs  | defaults,size=16M | 0 0 |
| tmpfs  | /dev/shm      | tmpfs  | mode=0777         | 0 0 |
| tmpfs  | /media/ram    | tmpfs  | defaults,size=16M | 0 0 |

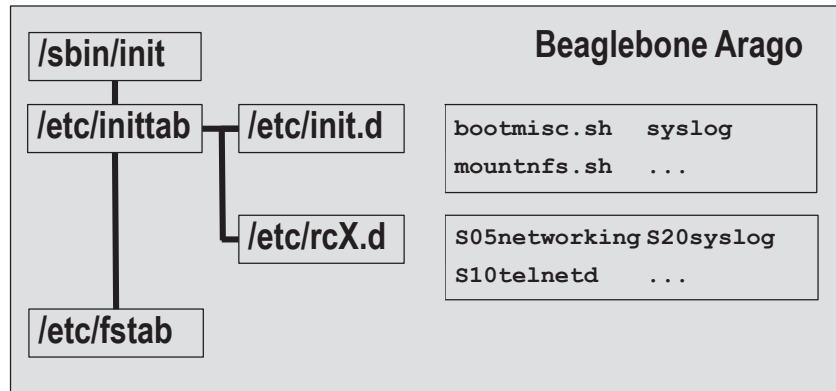
**# uncomment this if your device has a SD/MMC/Transflash slot**

**#/dev/mmcblk0p1 /media/card auto defaults,sync,noauto 0 0**

A similar startup configuration file is “/etc/fstab” (standing for FileSystem TABLE)

We will discuss the procedure of mounting block storage devices in module 05. Mounting a block storage device basically means gaining access to the data on the device. Any devices that are specified in “/etc/fstab” are mounted each time that the system starts up.

## /sbin/init vs distribution

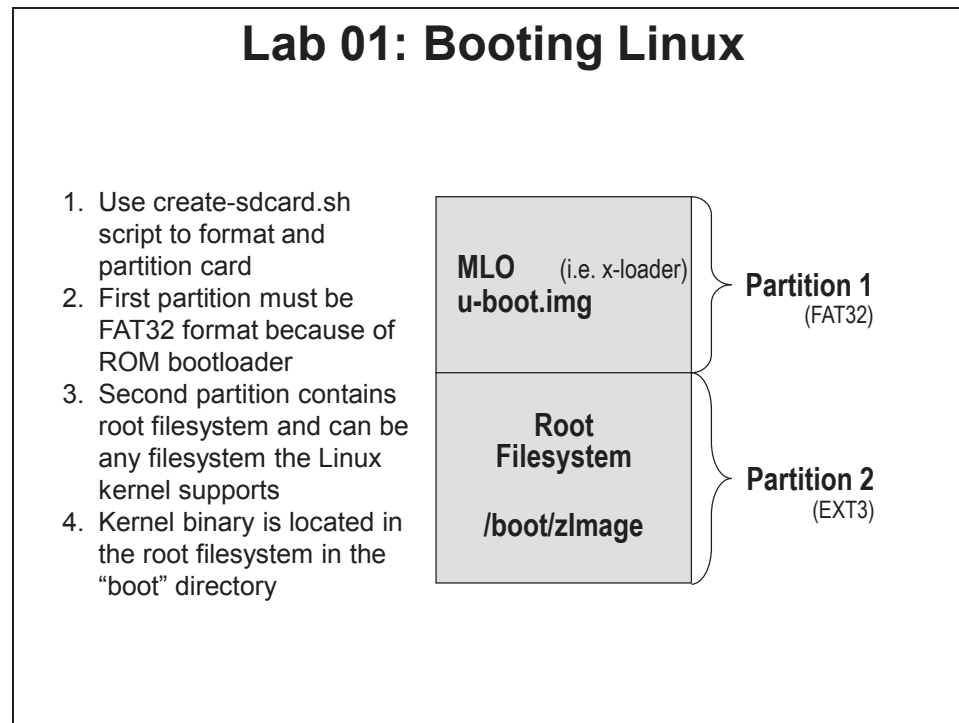


**Ultimately, the meaning of initialization scripts are driven completely by `/sbin/init`**

**Initialization scripts and tables only have meaning for a given distribution if recognized by `/sbin/init`**

This slide shows a summary of the startup process, startup tables and startup directories used on the Arago distribution as discussed on the previous slides.

## Lab 1 Introduction



In lab 01 you will use the “create-sdcard.sh” shell script from the Texas Instruments Software Development Kit (SDK) to format and partition a bootable SD card. This script will create a boot partition (partition 1) and a root filesystem partition (partition 2). The first partition must be formatted with the “FAT32” filesystem type (circa Windows 98) because this is the only filesystem that the ROM bootloader on the AM335x device can interpret. The second partition can be formatted with any filesystem type that u-boot and the Linux kernel can parse. A common Linux filesystem for embedded devices is the “ext3” filesystem.

# Module 02: Linux Distributions

---

## Introduction

A Linux distribution is the entire collection of software necessary to run Linux on a given platform. At the minimum, this is a bootloader, a Linux kernel and a Linux filesystem. Additionally, some distributions incorporate various tooling for configuring or debugging the distribution.







This module will discuss distributions generically, then focus on the Arago distribution, which is the distribution that is tested and maintained by Texas Instruments for use on TI platforms.

## Module Topics

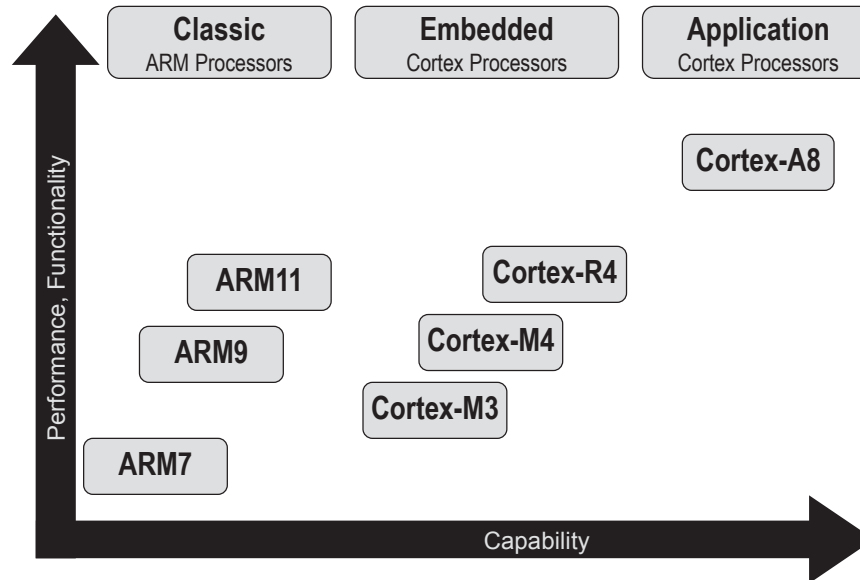
|   |             |
|---|-------------|
| <b>Module 02: Linux Distributions .....</b> | <b>2-1</b>  |
| <i>Module Topics.....</i>                   | <i>2-2</i>  |
| <i>(Short) Product Overview .....</i>       | <i>2-3</i>  |
| <i>Linux Terminal.....</i>                  | <i>2-7</i>  |
| <i>MMU and Dynamic Libraries.....</i>       | <i>2-16</i> |
| <i>Lab 02: Terminal Manipulation.....</i>   | <i>2-23</i> |

## (Short) Product Overview

### TI Embedded Processors Portfolio

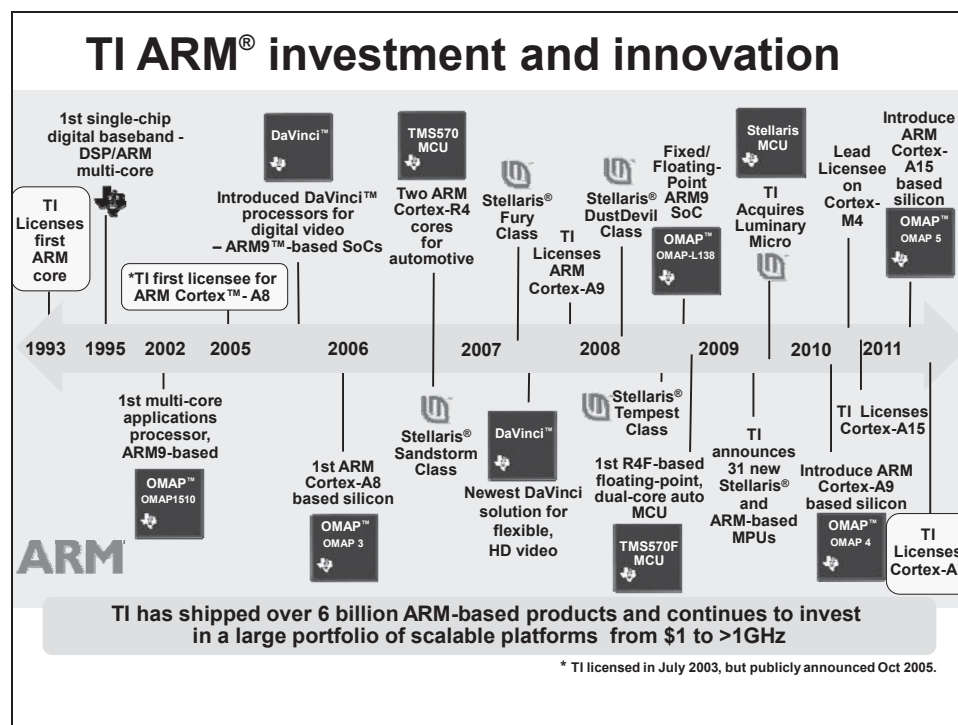
| Microcontrollers   |  |   | ARM-Based  |  | DSP   |
|--|--|---|--|--|---|
| 16-bit   | 32-bit Real-time   | 32-bit ARM  | ARM+   | ARM + DSP  | DSP   |
| <b>MSP430</b><br>Ultra-Low Power<br>Up to 25 MHz<br>Flash 1 KB to 256 KB<br>Analog I/O, ADC, LCD, USB, RF<br>Measurement, Sensing, General Purpose<br>\$0.49 to \$9.00 | <b>C2000™</b><br>Fixed & Floating Point<br>Up to 300 MHz<br>Flash 32 KB to 512 KB<br>PWM, ADC, CAN, SPI, I²C<br>Motor Control, Digital Power, Lighting, Sensing<br>\$1.50 to \$20.00 | <b>ARM</b><br>Industry Std Low Power<br><100 MHz<br>Flash 64 KB to 1 MB<br>USB, ENET, ADC, PWM, SPI<br>Host Control<br>\$2.00 to \$8.00 | <b>ARM9 Cortex A-8</b><br>Industry-Std Core, High-Perf GPP<br>Accelerators<br>MMU<br>USB, LCD, MMC, EMAC<br>Linux/WinCE User Apps<br>\$8.00 to \$35.00 | <b>C64x+ plus ARM9/Cortex A-8</b><br>Industry-Std Core + DSP for Signal Proc.<br>4800 MMACS/1.07 DMIPS/MHz<br>MMU, Cache<br>VPSS, USB, EMAC, MMC<br>Linux/Win + Video, Imaging, Multimedia<br>\$12.00 to \$65.00 | <b>C647x, C64x+, C674x, C55x</b><br>Leadership DSP Performance<br>24,000 MMACS<br>Up to 3 MB L2 Cache<br>1G EMAC, SRIO, DDR2, PCI-66<br>Comm, WiMAX, Industrial/Medical Imaging<br>\$4.00 to \$99.00+ |
|   |   |    |   |    |    |

### ARM CPU Processor Cores



<http://www.arm.com/products/processors/index.php>

TI's ARM core's supporting Linux...



## AM335x Cortex™-A8 based processors

### Benefits

- High performance Cortex-A8 at ARM9/11 prices
- Rich peripheral integration reduces complexity and cost

### Sample Applications

- Home automation
- Home networking
- Gaming peripherals
- Consumer medical appliances
- Printers
- Building automation
- Smart toll systems
- Weighing scales
- Educational consoles
- Advanced toys
- Customer premise equipment
- Connected vending machines

### Software and development tools

- Linux, Android, WinCE and drivers direct from TI
- StarterWare enables quick and simple programming and migration among TI embedded processors
- RTOS (QNX, Wind River, Mentor, etc) from partners
- Full featured and low cost development board options

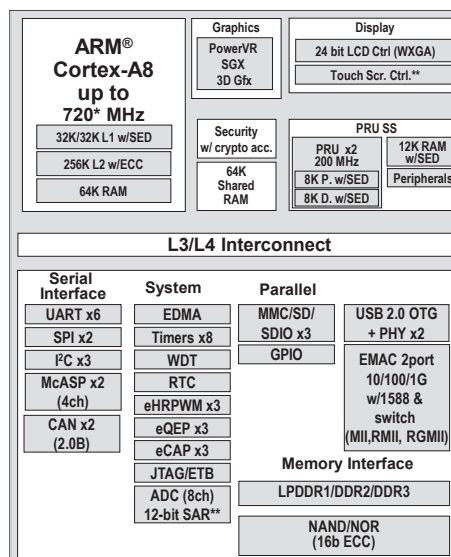
### Power Estimates

- Total Power: 600mW-1000mW
- Standby Power: ~25mW
- Deep Sleep Power: ~5-7mW

### Schedule and packaging

- Samples: October 31, 2011; Production: 2Q'12
- Dev. Tools: Order open October 31, 2011
- Prelim docs: available today
- Packaging: 13x13, 0.65mm via channel array 15x15, 0.8mm

Availability of some features, derivatives, or packages may be delayed from initial silicon availability  
Peripheral limitations may apply among different packages  
Some features may require third party support  
All speeds shown are for commercial temperature range only



\* 720 MHz only available on 15x15 package. 13x13 is planned for 500 MHz.  
\*\* Use of TSC will limit available ADC channels.  
SED: single error detection/parity



## AM335x/WL1271 Development Kit

AM335x EVM  
available for  
purchase through  
TI E-store:  
TMDXEVM3358



COM6M WL1271 Adapter Board  
is included in AM335x EVM Kit

### Kit Contents

- TI AM335x EVM Kit includes AM3358 Microprocessor, 512 MB DDR2, TPS65910 Power management IC, 7" LCD, Software & Tools, WL1271 COM6M Adapter Card

### Demos

- QT based WLAN and Bluetooth® demo Applications integrated into SDK
- WPA Supplicant and Host Apd GUIs for WiFi Station and Soft AP setup
- Profusion Bluetooth GUI with BMG (Scan, Pair, Connect), A2DP, FTP, OPP and SPP demos

### Software

- Open Source Linux 3.2 drivers
- Pre-Integrated with TI Linux SDK
- mac80211 Open Source WLAN Driver
- BlueZ Open Source Bluetooth® stack
- BlueZ OpenObex Profiles

### Documentation

- User and Demo guide, Releases: [TI Wireless Connectivity Wiki](#)
- [Module WL1271-Type TN Datasheet](#) from Murata

## WL1271-TypeTN (Murata)

WLAN 802.11 b/g/n and Bluetooth® v4.0 BLE Module



LBEE5ZSTNC-523

### Features

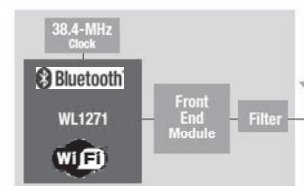
- IEEE 802.11 b/g/n compliant
- Bluetooth 4.0 with Bluetooth Low Energy
- Wi-Fi Direct
- TI's proven 6th generation Wi-Fi and Bluetooth solution
- Pre-integration with high performance Cortex-A8 based AM335x processor platform
- Open-source compliant Wi-Fi and Bluetooth drivers
- FCC Certified, ETSI & EMC Tested WL1271 module
- Sample applications and demos

### Applications

- Mobile consumer devices
- Industrial and home automation, metering
- Portable data terminals
- Video conferencing, video camera

### Benefits

- Seamless, direct and high throughput Wi-Fi connectivity between devices (no external access points needed)
- High throughput, reliable signal integrity, best in class coexistence, enhanced low power
- Simplified and reduced hardware and software integration effort, get started quickly
- Platform enables high performance processing and increased level of integration at value-line pricing
- Open-source compliant Wi-Fi and Bluetooth drivers
- Certified modules lowers manufacturing and operating costs, saves board space and minimizes RF expertise required



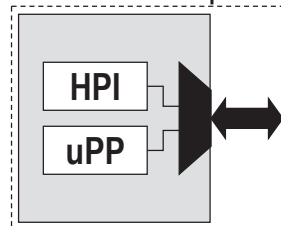
[www.ti.com/wl1271typetn](http://www.ti.com/wl1271typetn)

## What is Pin Multiplexing?

| Peripherals |         |         |        |
|-------------|---------|---------|--------|
| Serial      | Storage | Master  | Timing |
| McBSP       | DDR2    | USB 1.1 | Timers |
| McASP       | SDRAM   | USB 2.0 | Watch  |
| ASP         | Async   | EMAC    | PWM    |
| UART        | SD/MMC  | uPP     | eCAP   |
| SPI         | ATA/CF  | HPI     | RTC    |
| I2C         | SATA    | EDMA3   |        |
| CAN         | RAM     | SCR     | GPIO   |

| PRU<br>(Soft Peripheral) | Video/Display<br>Subsystem |
|--------------------------|----------------------------|
| CAN                      | Capture                    |
| UART                     | Analog Display             |
| What's Next?             | Digital Display            |
| DIY...                   | LCD Controller             |

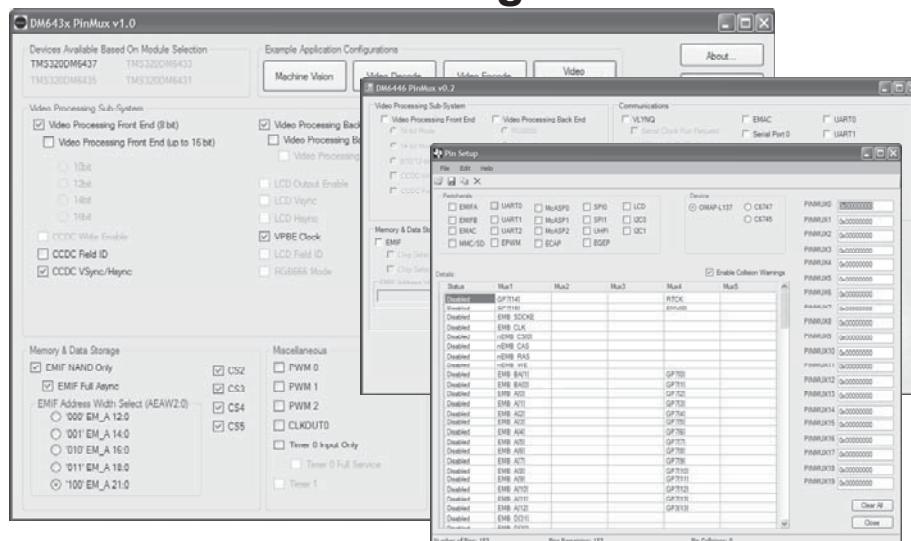
### Pin Mux Example



- ◆ How many pins is on your device?
- ◆ How many pins would all your peripheral require?
- ◆ Pin Multiplexing is the answer – only so many peripherals can be used at the same time ... in other words, to reduce costs, peripherals must share available pins
- ◆ Which ones can you use simultaneously?
  - Designers examine app use cases when deciding best muxing layout
  - Read datasheet for final authority on how pins are muxed
  - Graphical utility can assist with figuring out pin-muxing...

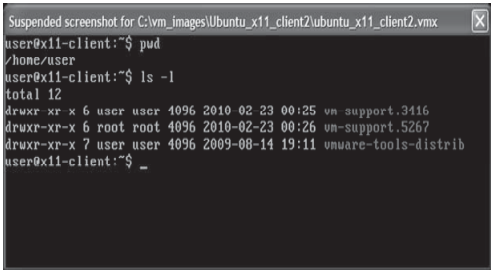
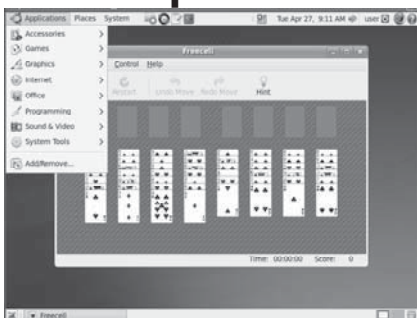
Pin mux utility...

## Pin Muxing Tools



# Linux Terminal

## Terminal vs. Desktop

- ◆ (Unlike Microsoft Windows) Linux is an inherently text-based operating system
- ◆ The graphics layers of Linux are simply applications that run on top of the native operating system
- ◆ There are multiple graphics layers to choose from including:
  - ◆ X11
  - ◆ qt/e
  - ◆ Wayland

One unique feature of Linux as an operating system is that it is fundamentally text-based, as opposed to the Windows operating system, which integrates graphics functionality at its core level. The primary interface to the Linux operating system is the text-based terminal as opposed to windows, which uses the graphics-based desktop as its primary interface.

Linux does provide support for graphics, but this graphics support runs as a layer on top of the Linux kernel (although this layer does interact with the FBDEV graphics driver that is a component of the Linux kernel.)

There are multiple graphics layers supported within the Linux community. Some of the most popular are X11, also known as XORG, Wayland, qt and qt/embedded. Because these layers are not integrated into the Linux kernel, any of the graphics layers (or even no graphics layer!) may be used in a given Linux distribution. Of course the choice of different graphics layers has a profound effect on the “look and feel” of the distribution.

## Basic Terminal Commands

|  |                                 |
|--|---------------------------------|
| <code>cat &lt;file&gt;</code>                  | dump file to terminal           |
| <code>cd &lt;directory&gt;</code>              | change working directory        |
| <code>cp &lt;source&gt; &lt;dest&gt;</code>    | copy a file                     |
| <code>cp -R &lt;source&gt; &lt;dest&gt;</code> | recursive copy                  |
| <code>pwd</code>                               | print working directory         |
| <code>ls</code>                                | list files in working directory |
| <code>mkdir &lt;directory&gt;</code>           | make a new directory            |
| <code>mv &lt;source&gt; &lt;dest&gt;</code>    | move a file or directory        |
| <code>rm &lt;file&gt;</code>                   | remove a file                   |
| <code>rm -R &lt;directory&gt;</code>           | recursive remove                |
| <code>rmdir &lt;directory&gt;</code>           | remove a directory              |
| <code>touch &lt;file&gt;</code>                | update the timestamp of a file  |

This slide shows a few of the most basic and commonly used Linux terminal commands with a brief explanation. For more information on any of these commands, you can use the “man” (MANual) command to print the manual page for a command by typing “man <command>” at the terminal. Additionally, most standard commands support a help feature. By typing the command followed by a “-h” flag and/or a “--help” flag, you can print the help.

## Terminal History

**Pressing the up arrow in a terminal will scroll through previously executed terminal commands**

**After scrolling to a command, you may use left and right arrows to edit the command and then return to execute the new command**

**<ctrl-r> brings up the “reverse-i-search” prompt allowing you to search the terminal history for a key phrase**

**You can execute the latest command beginning with “<string>” using “!<string>”**

The history capability of the terminal is a tool that will save users a significant amount of time. By pressing the up arrow, terminal users can scroll through previously executed commands. By selecting a command with the “enter” key, one of these previously executed commands can be re-run, and users can even edit the command before re-executing using the left and right arrows.

“Power” users may also be interested in the “reverse-i-search” capability. By pressing ctrl-r, a terminal user can search the terminal history for a key phrase. Even faster than this is the “bang” (i.e. exclamation key) followed by a string, which will execute the most recent command that begins with the specified string.

## Advanced Terminal History

Terminal history is stored in a file defined by the environment variable “HISTFILE” usually `~/.bash_history`

You may view previous commands using the “history” command.

You may execute a command N previous using “!*N*”

```
# history | tail
1    ls
2    cd ~/myfiles
3    cp this.txt that.txt
# !1
```

The terminal history is stored in a file defined by the “HISTFILE” environment variable, which may be set via many methods but is typically set within the hidden file “.bash\_history” at the top level of the user’s home directory.

This file can be viewed using the “history” command. This command also lists an index for each command which can be used with the “bang” notation to quickly re-execute the command. In the example shown below, the “ls” command from the history has index “1” and “!1” typed at the terminal will re-execute this command.

## Terminal Autofill

**You can have Linux autofill the name of a file or directory by entering the first few letters and pressing <tab>**

**If the letters you have entered are unique in terms of files in current directory, the file name will be completed**

**If multiple files match the letters you have entered, you may press <tab> twice in rapid succession and the terminal will print all files that match what you have entered.**

**Terminal autofill will save you a lot of time over the long run!**

Another powerful time-saving tool in the Linux terminal is the autofill capability. File names and commands (which are just executable files) can be more quickly entered by typing the first portion of the file name and pressing the “tab” key to have the terminal autofill the remainder.

If the first portion of the file name entered is sufficient to determine a unique file name, the entire file name will be filled in by pressing the “tab” autofill. If more than one file name matches the string entered, the terminal will autofill the file name up to the point of the first conflict. For instance if the directory contains “myfile1” and “myfile2” and you enter “my” and press “tab,” then the terminal will complete “myfile” but will not complete “1” or “2.”

In the instance of such a conflict, the user may press the “tab” key twice in rapid succession and the terminal will provide a list of all file names which match the text entered to that point.

## Linux Redirection

```
$ ls > lsout.txt
$ grep abc < lsout.txt
  abcfile.txt
```

The first command stores the output (stdout) of “ls” in a file `lsout.txt`

The second command uses `lsout.txt` as the input (stdin) to the program “grep”

```
$ ls | grep abc
  abcfile.txt
```

Piping the stdout of “ls” into the stdin of “grep” produces a similar result

A more advanced capability of the Linux terminal is redirection. Programs contain the concept of “standard input” and “standard output” streams. Normally a program that is executed from the terminal will attach its standard input to the terminal keyboard and any text output over the program’s standard output will be displayed on the terminal.

Using redirection, the user can redirect the standard output to a file or even use a file to provide the standard input to a program. In the example above, “`ls > lsout.txt`” will redirect the standard output of the “ls” command into a file named “`lsout.txt`.” In the second part of the example, the “`grep abc`” command, which will filter strings containing “abc” from the standard input, is provided with the “`lsout.txt`” file as its standard input, and so the command prints any line that contains the “abc” string from the file, in this case a single file that is named “`abcfile.txt`”

Another form of redirection, referred to as a pipe, attaches the standard output of one program to the standard input of another. As many programs as desired can be chained together using piping, creating a flow from one program to the next. In the final example above, the standard output of the “ls” command is redirected or piped to the standard input of the “grep abc” command using the “|” character. The effect is the same as in the previous example but does not require the intermediate “`lsout.txt`” file.



## Shell Scripts

**The shell program is responsible for executing text commands entered at the terminal prompt**

**It is located at /bin/sh which is generally a soft link to either /bin/bash or /bin/dash**

**The shell program can also interpret text files called shell scripts which contain a sequence of commands**

The Linux application which is used to execute the commands entered at the terminal is called the “shell.” There are a number of shells available in the Linux community, the most popular of which are the “Bourne Again SHell” (BASH) and its lighter-weight counterpart the “Debian Almquist SHell.” Linux uses the program located within the filesystem at the location “/bin/sh” to launch the terminal shell. Generally this location does not contain an actual program but rather a soft link to the actual terminal program.

In addition to interpreting and executing commands from the terminal, the shell program can be used to interpret and execute commands from a text file, often referred to as a “shell script.” Often these shell scripts are named with a “.sh” extension, but this is not required. Any text file can be executed as a shell script if it is marked with executable permission.

## BASH Shell Scripts – Variables

### terminal

```
$ gedit myscript
$ chmod a+x myscript
$ ./myscript arg1 arg2
    arg1
    prearg2post
    myvalue
```

### myscript

```
#!/bin/bash
MYVAR="myvalue"
echo ${1}
echo pre${2}post
echo ${MYVAR}
```

**The parenthesis are not required for variable expansion but help when appending or prepending to a string.**

A shell script may be as simple as a direct capture of commands as they would be typed on the terminal, but there are additional tools that may be employed within the script. One of the simplest, but quite helpful, of these tools is variable expansion. In the example above, a file named “myscript” is edited with the “gedit” program.

Any line of a shell script that begins with the hash (“#”) character is a comment. The myscript file in this example begins with a special comment, “#!/bin/bash” that specifies the script is to be executed with the shell program located at “/bin/bash.” This comment is not required, and if not provided the system will use the default shell located at “/bin/sh” but as different shells have subtle differences it is generally recommended that shell scripts begin with this declaration.

The next line of the script declares the variable “MYVAR” and initializes it with the value “myvalue.” In this example the variable is named with all capital letters. This capitalization is not required but is a standard style guideline used in most scripts for readability.

The next three lines show examples of dereferencing variables. A variable is dereferenced using the dollar character, “\$.” It is not required to bracket the variable name, i.e. both “\$MYVAR” and “\${MYVAR}” will dereference the variable, but brackets allow text to be prepended or appended to the variable.

The example above also shows the special variables “\${1}” and “\${2},” these variables refer to the first and second parameter provided to the script, respectively. “\${0}” is another special variable that contains the calling function, in this case the name of the script itself, as called from the command line.

## BASH Shell Scripts – Functions

### myscript

```
#!/bin/bash  
  
echo $(cat ${0})
```

### terminal

```
$ ./myscript  
#!/bin/bash echo $(cat ${0})
```

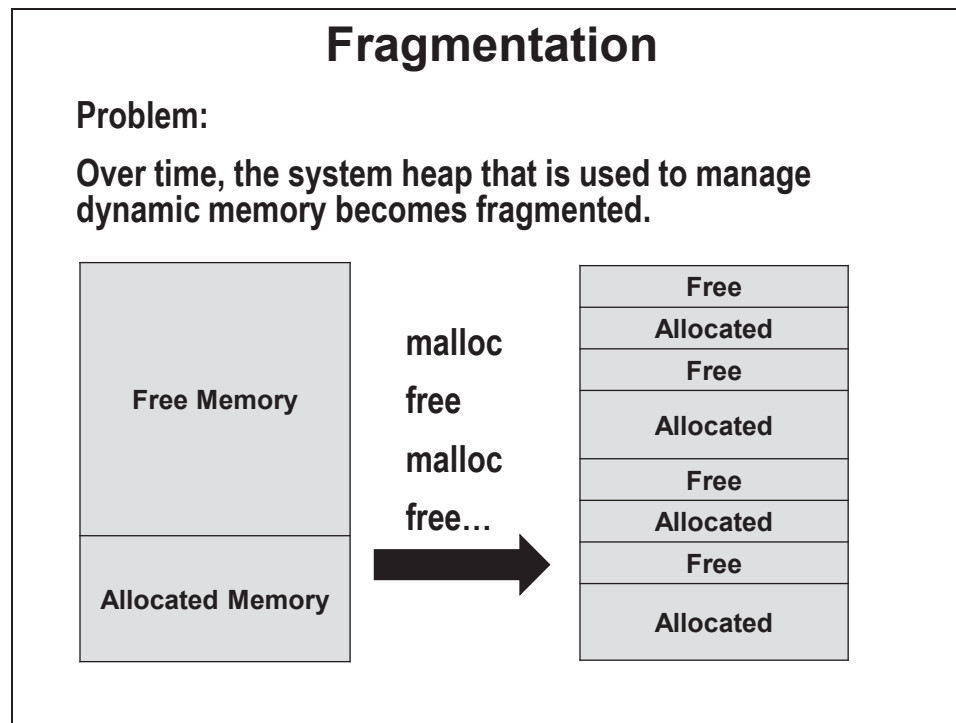
**Parenthesis evaluate a command and insert the result inline**

Commands may also be inserted in-line to the script. Generally a command is entered on its own line of the script, which simply executes that line in the shell. If instead the command is encased in “\$( )” the command is still executed in the shell but instead of its output being displayed on the terminal, its output will be inserted in-line into the script.

In the example above, the output of the “\$(cat \${0})” command is inserted into the script so that it becomes the parameter to the “echo” command, which then echoes it to the terminal.

Since the \${0} refers to the calling function, which is just the name of the script, and since “cat” is a utility that sends the contents of a text file to standard out, the contents of this script are inserted in-line to the echo command so that the end effect of the script is to dump the contents of itself to the terminal.

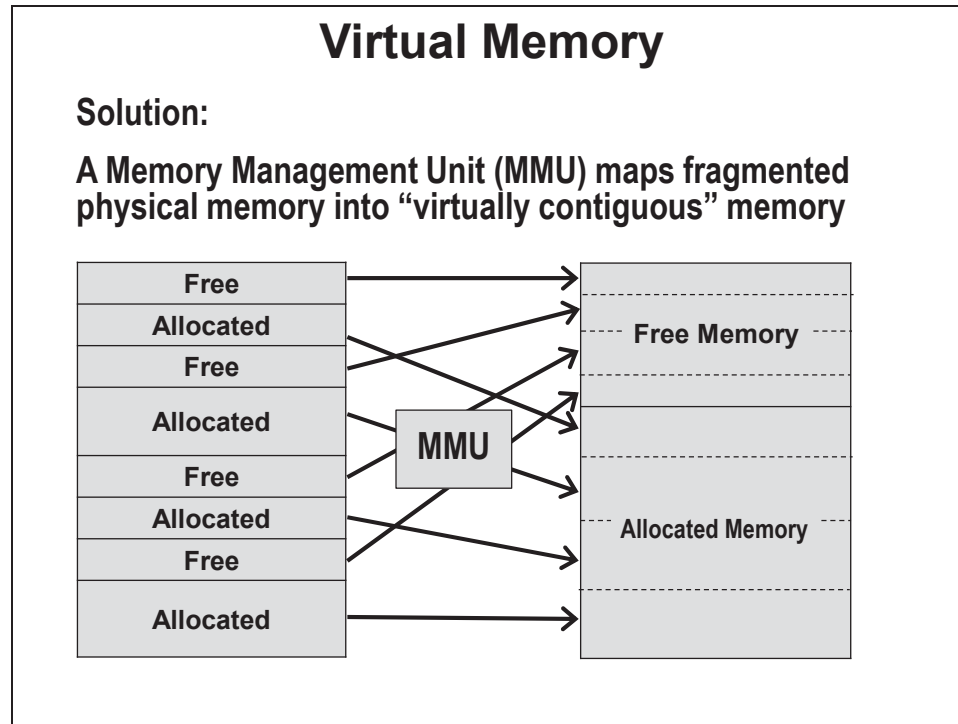
## MMU and Dynamic Libraries



A problem that occurs over time as a result of the allocation and de-allocation of dynamic memory from a heap is fragmentation. Over time the heap memory becomes fragmented, which is to say that the allocated memory is intermixed with free memory. The issue with fragmentation is that most programs require memory to be allocated in continuous blocks, i.e. arrays. Thus, as memory becomes increasingly fragmented, the largest size of continuous memory, which is the usable memory, becomes increasingly smaller, even if the total sum of free memory remains constant. For a system that is required to run for weeks, months, even years, this fragmentation would eventually cripple the system to where it could no longer run efficiently.

As a side note, a related form of fragmentation occurs on hard-disk drives. Users of Windows 98 might remember running the disk defragmenting utility a few times every year to clean up this fragmentation. (Later versions of Windows defragment the disk automatically so that it is not necessary to run this utility.)

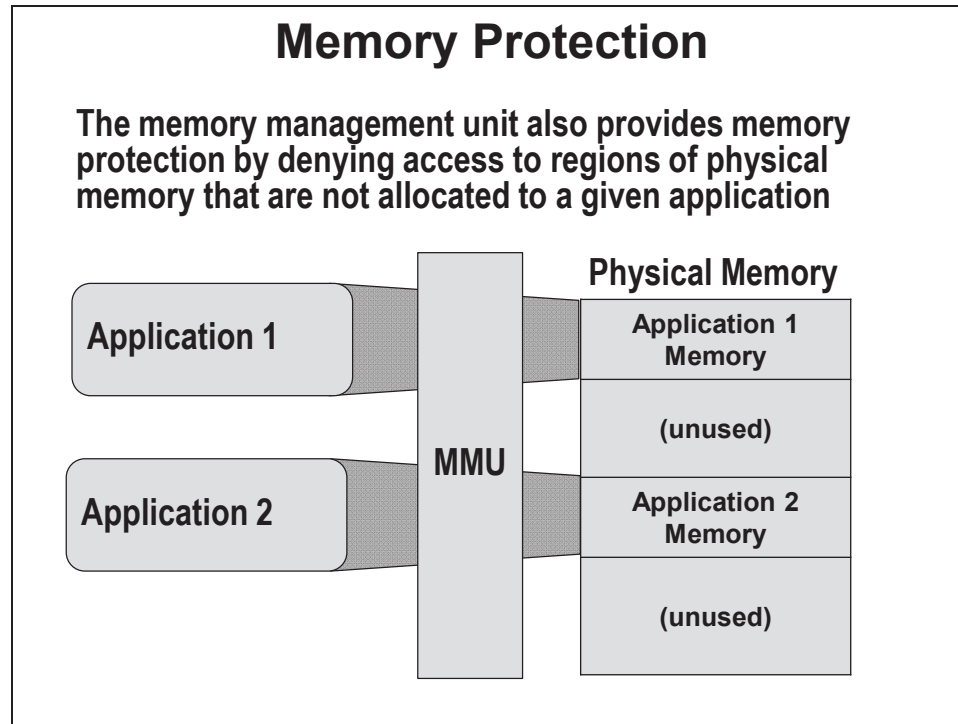
What we are concerned with in this example, however, is fragmentation of the RAM memory. Allocation and de-allocation of memory in RAM generally occurs much more frequently than allocation and de-allocation of files on a hard disk drive, so that fragmentation in RAM becomes an issue for a system much more quickly.



One of the most common solutions to the issue of fragmentation in RAM memory is to reorganize physical memory into a set of virtual memory (using virtual addressing) via a hardware block called the “Memory Management Unit” or MMU for short.

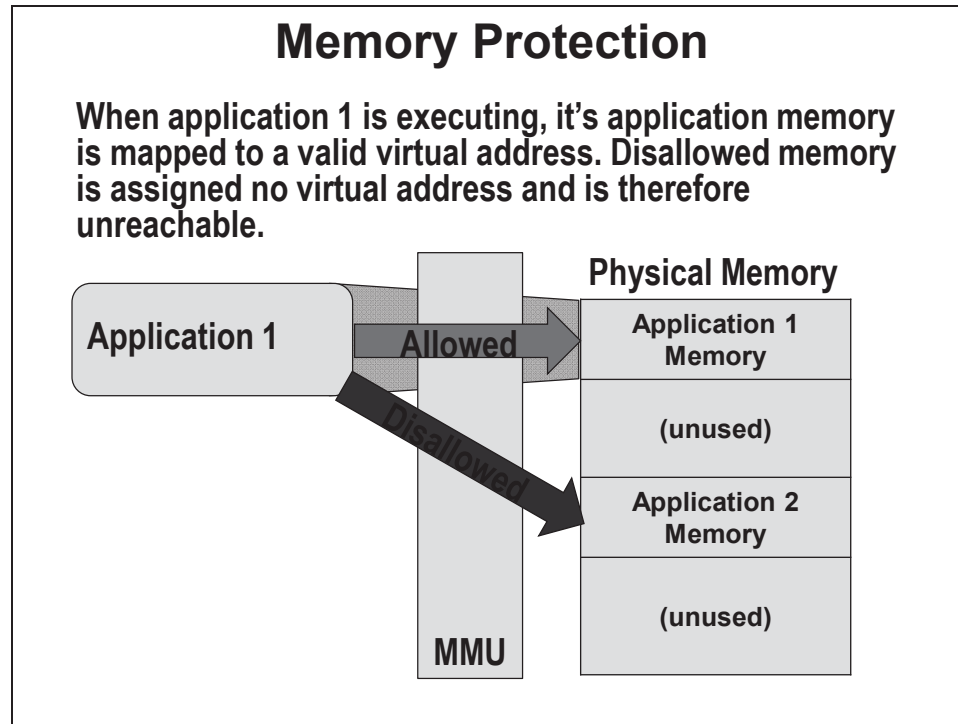
The MMU reorganizes the physical addresses of memory segments into virtual addresses by applying individual offsets to each block. This provides a set of virtual addresses in which the free memory is grouped together into one continuous block. The advantage of using virtual addresses instead of moving the blocks of memory with a copy operation is that it is much faster from a processor cycle standpoint.

Every time that the processor core accesses memory over one of its program or data buses, the address of the memory accessed is translated by the MMU. This virtual to physical translation is inserted into the processor pipeline on modern processors so that it adds no overhead to memory accesses.



Another capability of the MMU is to provide memory protection. Every memory access made by the processor core is required to go through the MMU for virtual to physical address translation. The MMU may then deny access to protected memory ranges, returning a “Segmentation Fault” event instead of allowing a read or write of the memory the processor is attempting to access.

By programming the MMU each time a context switch is made between the various applications (also known as “processes” on the Linux operating system) the MMU is reprogrammed with the allowed and denied memory ranges associated with that given process. This provides hardware memory protection between applications, allowing the Linux operating system to assign ranges of memory to each application which cannot be accessed by other applications on the system.

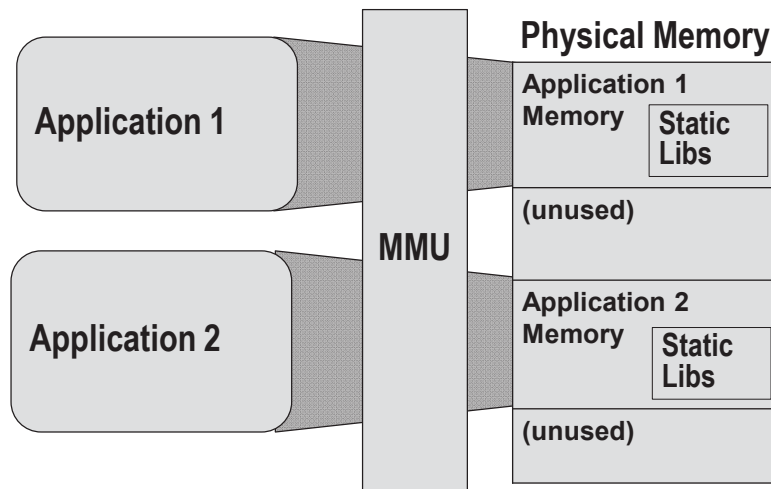


Thus, when the OS is within the context of Application 1, the MMU is programmed to allow access to the memory that has been assigned to the application, but if the application attempts to access memory outside of this range, such as the memory associated with Application 2, the access will be denied.

Additionally, the Linux kernel automatically protects its own program and data memory from the other programs running on the system. This provides robustness, and, at least in theory, guarantees that the OS will not crash as a result of improper memory handling by one of the applications running on the system. Note that most distributions of Linux require MMU hardware in order to execute and that processors which do not have hardware MMU support cannot run the Linux operating system.

There are special distributions of Linux, often termed “microcontroller Linux” which do not require an MMU. These distributions generally have a heavily reduced feature set, however.

## Static Linking and .a Libraries (Archives)



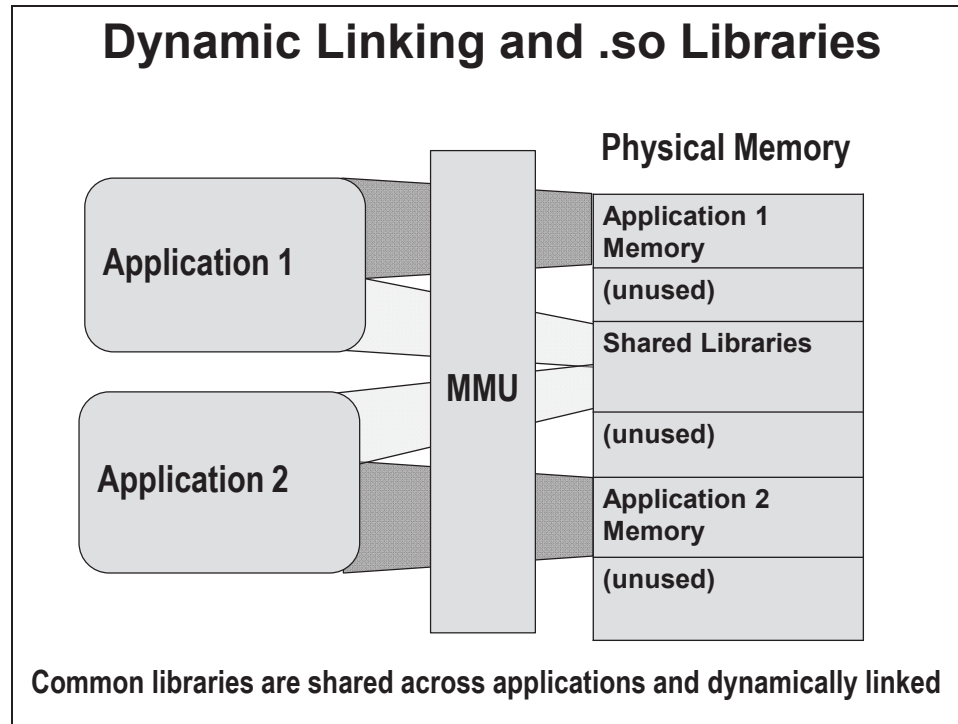
**Common libraries are shared across applications and dynamically linked**

One of the characteristics of Linux, by design, is that the operating system utilities are implemented via a large number of small, simple applications. When a command is typed into the Linux terminal, the main interface into the O/S, that command is executed as an executable file or program. Most Linux distributions implement each command as a separate application program, leading to a large number of applications on the system. The exception to this is in distributions that use a program called “busybox” to handle the terminal commands. Busybox is a single application that can interpret many of the common Linux terminal commands. Busybox uses fewer cycles and far less system memory than the conventional approach of implementing each command in a separate application, although not all of the features of these commands are implemented. Because of this efficiency, busybox is commonly used in distributions targeted for use in embedded systems.

Even if busybox is used, however, there are still a number of individual programs on the Linux system. These individual programs all use function calls from the basic Linux system libraries. For instance, nearly every program on a Linux system uses the “printf” command from the Standard Input/Output library (stdio) to display text on the terminal.

If every program on the system were to statically link the printf code from the stdio library into its program, the program code for this function would be repeated in separate copies embedded into each program that calls it.





So for a system that has hundreds of individual programs, it would be horribly wasteful to statically link system libraries into each program. Instead, Linux systems generally link these programs against shared libraries. With shared libraries, the function referenced from the shared library is not embedded into the program, instead it is left as a reference into a shared library. When the application is launched at run time, this reference is read by the “ld” loading utility and the reference is dynamically linked to the corresponding shared library call. This allows a single copy of the function to be shared across multiple applications. Note that the MMU only maps the shared library function across multiple applications. The remainder of the application space remains protected.

Dynamic linking is handled by the ld program, and most users will not need to know the details of how it is accomplished. In fact most new users are not even aware that programs they are launching from the terminal are dynamically linked.

## Linking Linux Libraries

```
# gcc myfile.c -lm -lpthread
```

This links the dynamic libraries:

**libm.so** – shared math library

**libpthread.so** – shared POSIX thread library

or the static libraries (archives)

**libm.a** – static math library

**libpthread.a** – static POSIX thread library

**Depending on which library appears first in the library search path.**

Dynamic or static linking of library functions into an application occurs at the linker phase of the gnu compiler chain (gcc.) Libraries are linked into the application using the “-l” flag. All libraries begin with the text “lib” and end with either an extension of “.a” for archive, which are static libraries, or “.so” for shared object, which are shared libraries. Note that when a library is linked this prepended “lib” and the extension are not specified, so to link the math library, “libm.so” specify “-lm”

By default the linker will attempt to link against the shared version of the library and only uses the static version of the library if the shared library is not found. This behavior can be overridden by using the “--static” flag which will link only against the static versions of each library.

## Lab 02: Terminal Manipulation

### Lab 02 – Beginner and Advanced Tracks

#### Beginner

(beginner) Basic Terminal Manipulation

(beginner) Basic File Manipulation

#### Advanced

(advanced) Linux Redirection

(advanced) Linux Scripting

Beginner ~ 10 minutes

### Should I take the Beginner Track?

#### Commands Covered:

```
cat cd cp pwd ls mkdir  
mv rm rmdir touch
```

#### Concepts Covered:

wildcard (\*)            home directory (~)  
autofill (<tab>)        terminal history (arrow keys)  
text editor (gedit)

(page intentionally left blank)

# Module 03: Linux Distributions

---

## Introduction

A Linux distribution is the entire collection of software necessary to run Linux on a given platform. At the minimum, this is a bootloader, a Linux kernel and a Linux filesystem. Additionally, some distributions incorporate various tooling for configuring or debugging the distribution.

This module will discuss distributions generically, then focus on the Arago distribution, which is the distribution that is tested and maintained by Texas Instruments for use on TI platforms.

## Module Topics

|  |  |
|--|--|
| <b>Module 03: Linux Distributions .....</b>  | <b>3-1</b>   |
| <i>Module Topics.....</i>                    | <i>3-2</i>   |
| <i>Linux Distributions .....</i>             | <i>3-5</i>   |
| <i>Linux Application Development.....</i>    | <i>3-13</i>  |
| <i>Bitbake Application Build .....</i>       | <i>3-17</i>  |
| <i>Lab 3: Open Embedded Hello World.....</i> | <i>3-27</i>  |
| <i>Extra Slides .....</i>                    | <i>Error! Bookmark not defined.-Error! Bookmark not defined.</i> |

**Q: I found a website that says GNU Linux supports \_\_\_\_\_, why isn't it in TI's (Arago) distribution?**

**A: GNU is a community of software developers, not an Operating System. Various Linux distributions (including Arago) are built by combining a subset of software available from the GNU community.**

A common source of confusion with new users of the Linux operating system occurs when a feature, such as a terminal command or a configuration file, is supported in one version of Linux and not in another. Often these users will find instructions to accomplish a given task only to find that partway through the instructions they are told to execute a command or modify a configuration file that does not exist on their system.

The reason this occurs is that GNU (which stands for “Gnu is Not Unix”) is a community of developers who develop various bodies of software under the GNU public licenses, and not a community of developers who develop a single operating system as a whole or atomic unit.

That said, GNU software is most commonly used in Linux systems, is generally developed for the purpose of being used in Linux systems. Also, there are contributors within the community who take it upon themselves to group the various pieces of GNU software together into a Linux distribution. However, there are many different Linux distributions, which are comprised of different combinations of GNU Public Licensed (GPL) code, and in some cases these different distributions have a dramatically different “look and feel.” In fact it is theoretically possible that two Linux distributions could share no common code except for the Linux kernel itself, which there is only one version of, and which is a minimum requirement for a distribution to be considered a “GNU Linux” Distribution.

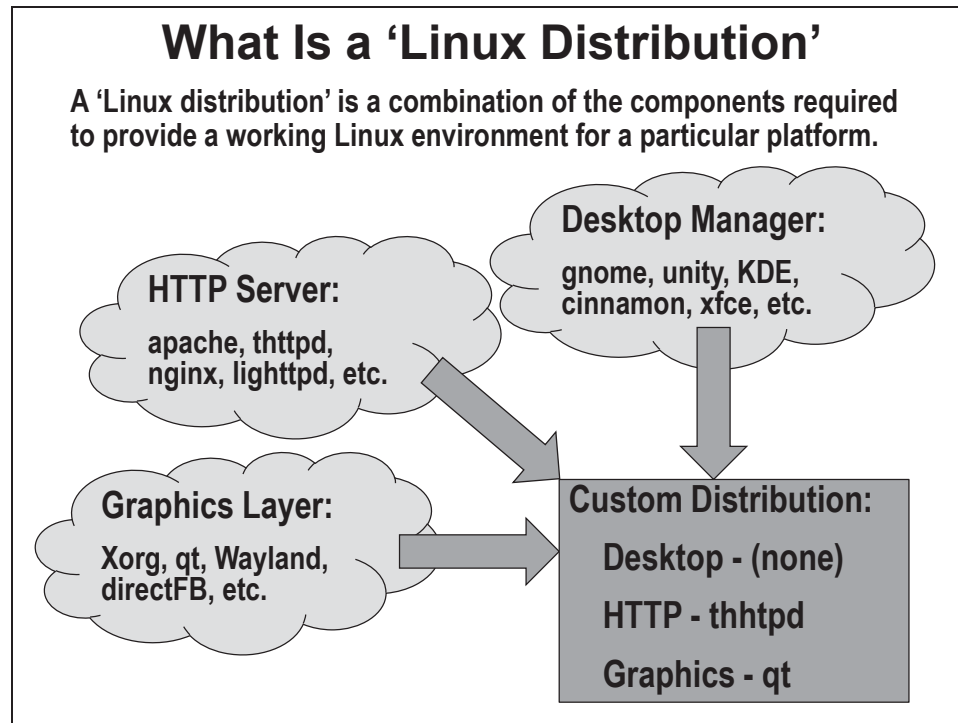
**Q: What I meant was: I need \_\_\_\_\_, which is available in the GNU Linux community, so how do I get it?**

**A: You can move to a different distribution that supports \_\_\_\_\_, or you can add it to Arago by compiling source using TI's SDK or by using OpenEmbedded.**

Once it is understood that different Linux distributions support various subsets of software from the GNU community, the next question is often how a person can add a feature or program that is available within the community to the distribution that they are using. This will be the subject of this chapter, which introduces various methods of either creating a new, custom distribution or of modifying one that already exists.



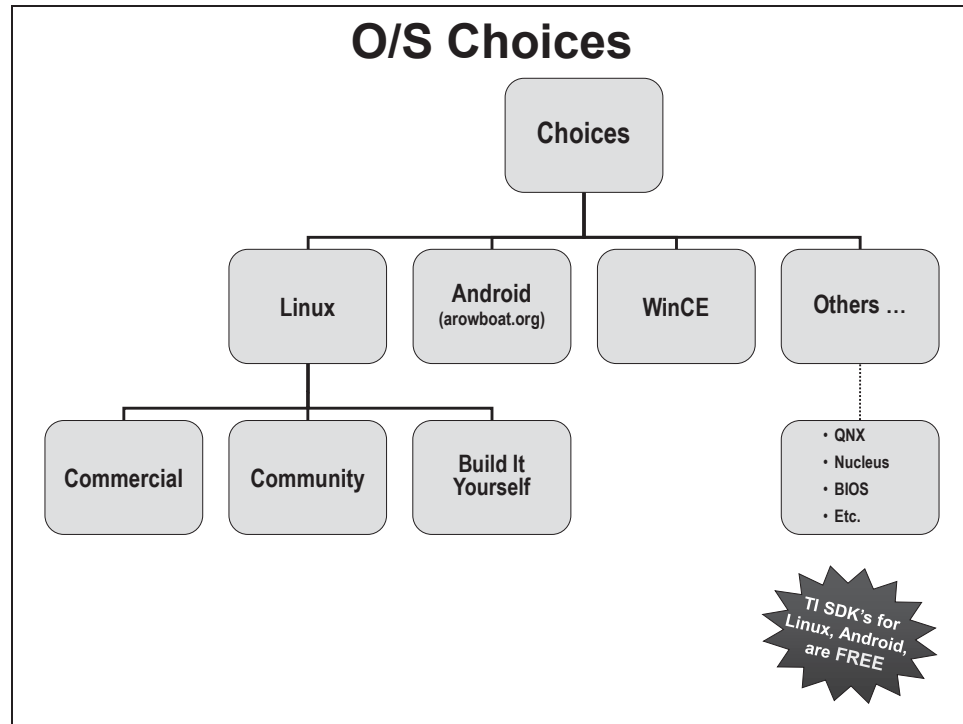
# Linux Distributions



There are various fundamental components that make up most high-level operating systems. For instance, most Linux distributions targeted towards desktop systems support a graphics layer for a desktop display, and nearly any modern system supports networking tools such as a HTTP server.

There are generally a number of options available within the community for accomplishing a given task. For instance, the most commonly used graphics layers are “XORG” “qt” and “Wayland.” Different graphics layers have different features. “XORG” is fully network transparent, meaning that it natively supports exporting a desktop across the network, but is heavy in terms of its resource usage. qt, particularly “qt/e” (qt for embedded) is much lighter weight than XORG and is very popular in embedded systems. Wayland, which is a more recent development, is a good middle ground between the two.

A distribution is formed by selecting different software packages from the community and bundling them together into a fully functional operating system. Because there are so many possibly combinations of software that might be selected from the community there are hundreds of different distributions of Linux available, some of which are very distinct in terms of their “look and feel.”



The first operating system choice a user of the ARM Cortex would make is not which Linux distribution to use, but actually which operating system to use. There are many operating systems supported on the ARM Cortex devices, including the three high-level operating systems: GNU Linux, Android and WinCE. Also, a number of Real-time Operating Systems such as QNX, Nucleus and TI-RTOS are available.

Assuming the user selects GNU Linux, there is still the choice of which distribution to use, and, at a higher level, if this distribution should be chosen from a commercial vendor, from within the open-source community or even if the user would like to assemble their own custom Linux distribution.

## Commercial O/S Vendors

### ◆ Linux

- MontaVista
- TimeSys
- Wind River
- Mentor
- Ridgerun

### ◆ WinCE

- Adeneo
- Mistral
- MPC Data
- BSQUARE

### ◆ RTOS

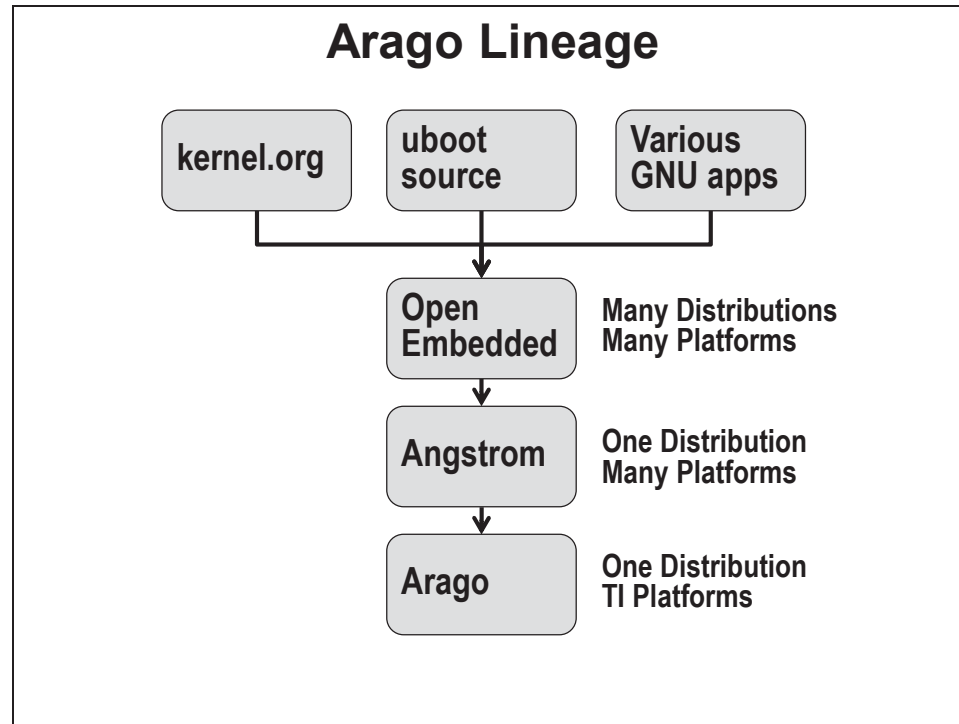
- Green Hills
- Wind River (VxWorks)
- ELogic (ThreadX)
- QNX
- Mentor (Nucleus)

### Linux Partner Strategy

- ◆ **Commercial:** provide support, off-the-shelf Linux distributions or GNU tools
- ◆ **Consultants:** provide training, general embedded Linux development expertise, or specific expertise for developing drivers or particular embedded applications
- ◆ [http://www.tiexpressdsp.com/index.php/Linux\\_Consultants\\_and\\_Commercial\\_Linux\\_Providers](http://www.tiexpressdsp.com/index.php/Linux_Consultants_and_Commercial_Linux_Providers)

Here is a partial list of third parties that provide Linux, WinCE and various RTOS solutions for TI's Arm-Cortex A8, A9 and A15 products.

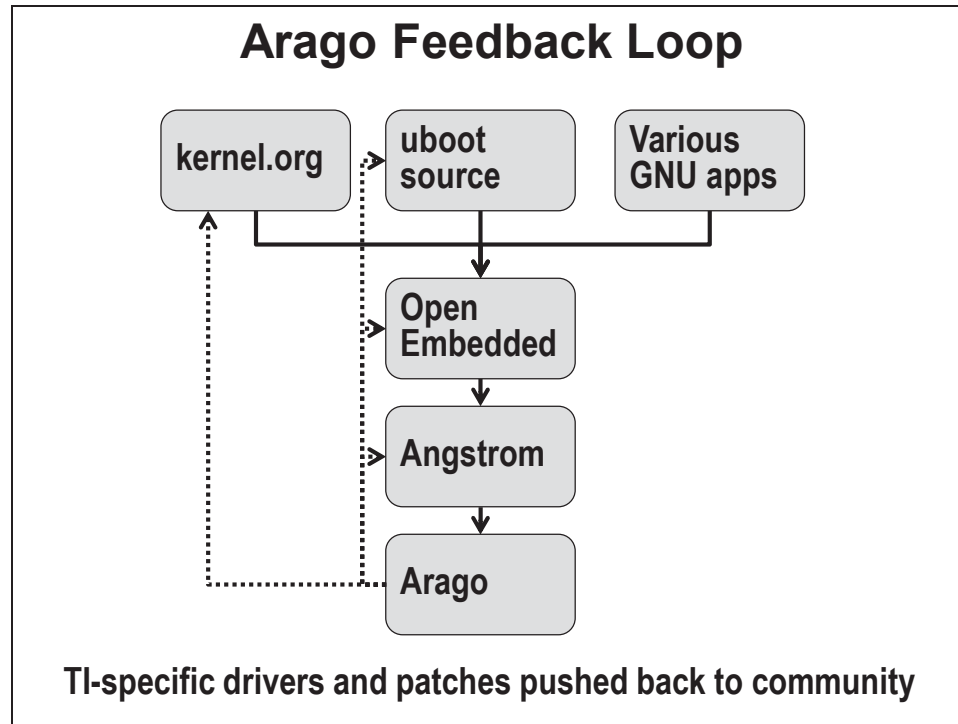
Commercial Linux vendors bring a high degree of Linux experience and support capability; however, they do charge for their services. Often, however, the amount of money that a customer spends with a third-party provider provides later savings in the form of reduced time to market.



Community Linux refers to any distribution of Linux that is provided free within the community. Most users of TI’s ARM devices select the Arago distribution, which is a community distribution that was developed by Texas Instruments and continues to be maintained and updated by TI. This distribution is heavily tested on TI devices.

Like most Linux distributions, Arago was not developed from scratch but was evolved from a pre-existing Linux distribution. In the case of Arago, this parent was a distribution named “Angstrom” from within the “Open Embedded” community. Open Embedded, which has now merged with the Yocto project, is an initiative designed to build entire Linux distributions from source code using recipe files and a tool named “bitbake.”

Within the Open Embedded project, there are a number of different distributions which are supported across a wide array of platforms. Angstrom is just one of those distributions, though this distribution is supported across many platforms. The Arago distribution is Open Embedded and Yocto-compliant and, like its parent distribution, can be built in its entirety from recipe files that are available in the Open Embedded community. This distribution is open source; however Texas Instruments only tests the distribution on TI-supported platforms.



All applicable code that is developed for the Arago distribution is fed back into the community, so that TI's Arago distribution is not an end-point or simple port of Linux to TI platforms, but rather Linux-community-wide development with Texas Instruments contributing heavily to the open-source community's development effort.

The drivers that TI develops to support TI platforms are contributed back to kernel.org, the definitive repository for the Linux kernel. Thus, a user doesn't have to download the Linux kernel from the Arago repository in order to have support for TI platforms. One can download the kernel from kernel.org or one of its mirrors and it will have full support for TI platforms.

## Arago Project – Your Best OE Starting Point

<http://arago-project.org/git/> edit

There are few git project contexts on this server.

Main Arago context: <http://arago-project.org/git/>

Arago Related Projects: <http://arago-project.org/git/projects>

Personal Work Areas: <http://arago-project.org/git/people>

| Project           | Description  | Owner             | Last Change  |
|-------------------|--|-------------------|--|
| arago-bitbake.git | Arago version of Bitbake build tool                                | Denys Dmytriyenko | 16 months ago <a href="#">summary</a>   <a href="#">history</a>   <a href="#">log</a>   <a href="#">tree</a> |
| arago-oe-dev.git  | Package build recipes; Snapshot of OpenEmbedded development branch | Denys Dmytriyenko | 13 days ago <a href="#">summary</a>   <a href="#">history</a>   <a href="#">log</a>   <a href="#">tree</a>   |
| arago-utils.git   | Arago-specific utilities and tools                                 | Denys Dmytriyenko | 4 weeks ago <a href="#">summary</a>   <a href="#">history</a>   <a href="#">log</a>   <a href="#">tree</a>   |
| arago.git         | Package build recipes; Arago specific or modified                  | Denys Dmytriyenko | 13 days ago <a href="#">summary</a>   <a href="#">history</a>   <a href="#">log</a>   <a href="#">tree</a>   |
| meta-arago.git    | Arago metadata layer for TI SDKs                                   | Denys Dmytriyenko | 3 weeks ago <a href="#">summary</a>   <a href="#">history</a>   <a href="#">log</a>   <a href="#">tree</a>   |
| meta-ti.git       | BSP layer for Texas Instruments platforms                          | Denys Dmytriyenko | 11 days ago <a href="#">summary</a>   <a href="#">history</a>   <a href="#">log</a>   <a href="#">tree</a>   |

txt opml

**Arago install and updates are distributed via git**

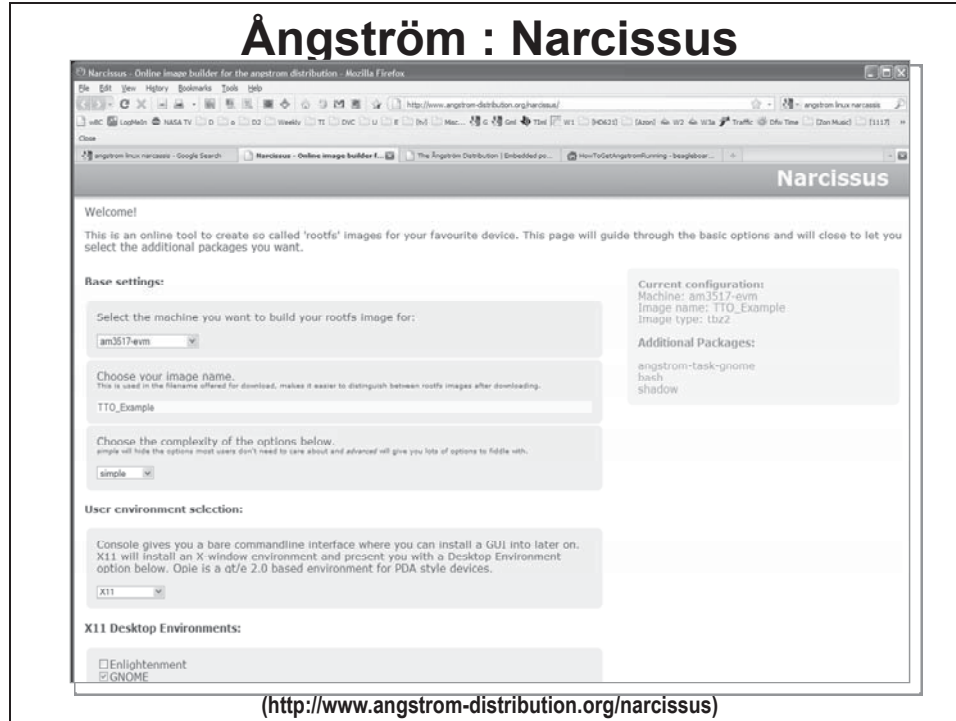
**Arago install includes Bitbake, Open Embedded recipes**

**A separate page contains pre-built binaries (snapshots)**

The best starting point for developers who wish to use open-source Linux on a TI platform is the Arago project website. As previously mentioned, TI pushes support for TI platforms back to the community including kernel.org; however, there is a time lag between when these patches are submitted and when they are accepted, so users who wish to have the absolute latest versions of software supported on TI platforms are encouraged to start with the Arago project.

The Arago project maintains the distribution as an open-source Open Embedded project. The recipe files used to build the distribution are freely available for download, so that users who wish to can rebuild the entire distribution from source. The project page also maintains snapshots of the distributions for those who would like to download a prebuilt version instead of building the distribution themselves.

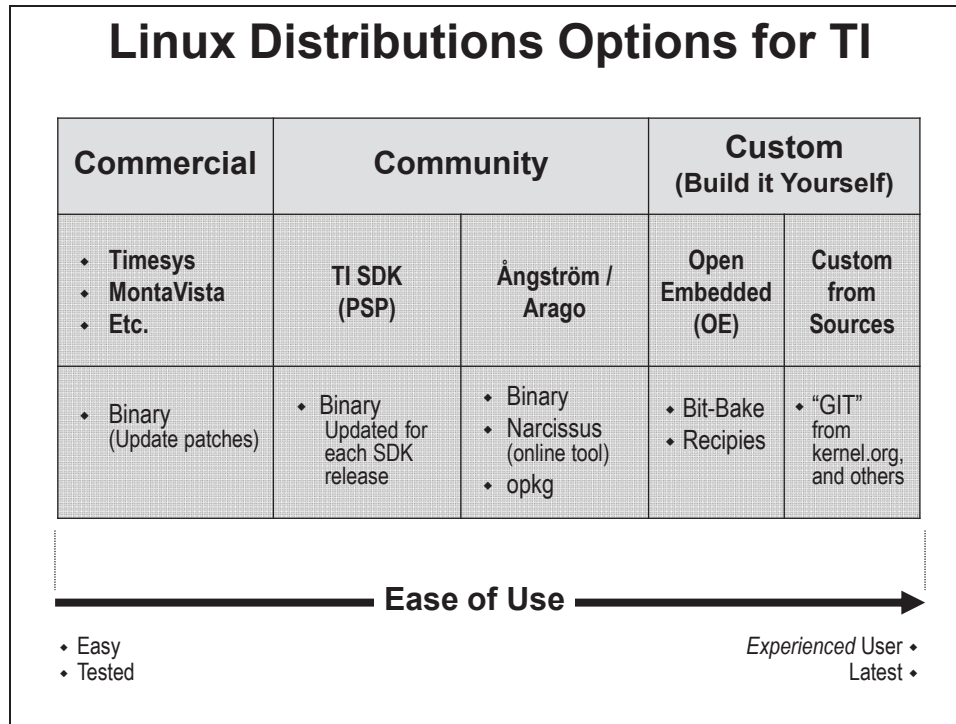
The Arago distribution recipe files also provide an excellent starting point for those users who wish to build their own custom distribution. It should be pointed out that this is not a trivial effort, but for those users wishing to undertake the effort, the Arago distribution provides a known-good starting point for development on TI platforms that is (relatively) easily modified using the Open Embedded bitbake recipe structure.



An even simpler method for creating a custom distribution than modifying the OE recipe files of the Arago distribution is to use a distribution build tool called “narcissus” from the Angstrom community.

Note that this tool is maintained by the Angstrom community and not by TI, and it is likely to import code and packages that have not been tested by TI on TI’s platforms. That said, since Arago is a derivative of the Angstrom distribution, there is a good chance that the open-source packages imported by the Narcissus tool and compiled for a TI platform will in fact work as expected.

The Narcissus tool is a graphical tool and a bit simpler and faster to use than modifying Open Embedded recipe files from the Arago distribution. Because it is graphical, it provides a more limited set of options to choose from than the bitbake recipe files, but many users find the granularity of choices to be more than sufficient.



Here we see a summary of the various options available for selecting or building a Linux distribution for a given project. They are organized by the amount of time and effort generally required to achieve a working system.

On the left-hand side is shown the option of purchasing support for a tested third-party distribution. This is little effort, but is also the only option shown that requires the user to spend money.

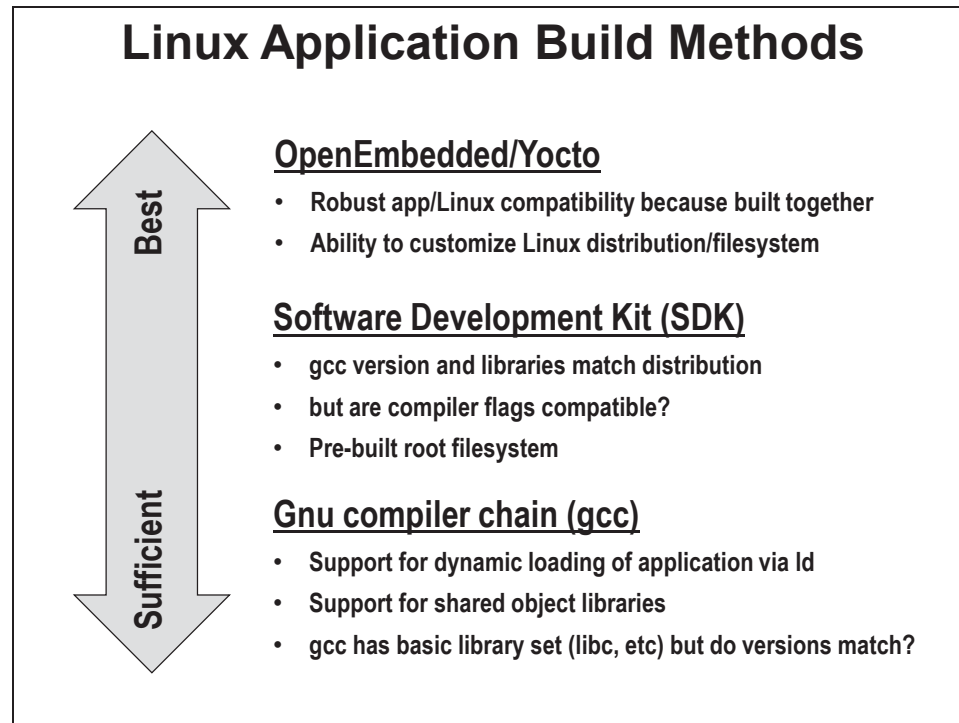
A common choice for many users is to use the Arago distribution out of the box (unchanged) from one of TI's software development kits (sdks.) Many users find the Arago distribution to contain all of the features that they require without modification, and since it is available without charge and is updated and maintained by Texas Instruments for TI platforms, it is a natural choice.

Those users who find that the out-of-the-box release of the Arago distribution does not fully meet their needs may want to investigate opkg – a package management tool that can be used to add packages from the Angstrom community on top of the Arago distribution. A similar technique would be to use the Narcissus tool to create a custom distribution using Angstrom community components.

Those users who would like the absolute most flexibility in designing and maintaining their own custom distribution may wish to investigate using the bitbake tool and Open Embedded recipes to do so. If so, the open-source recipe files of the Arago distribution, which is built using bitbake, provide an excellent starting point.



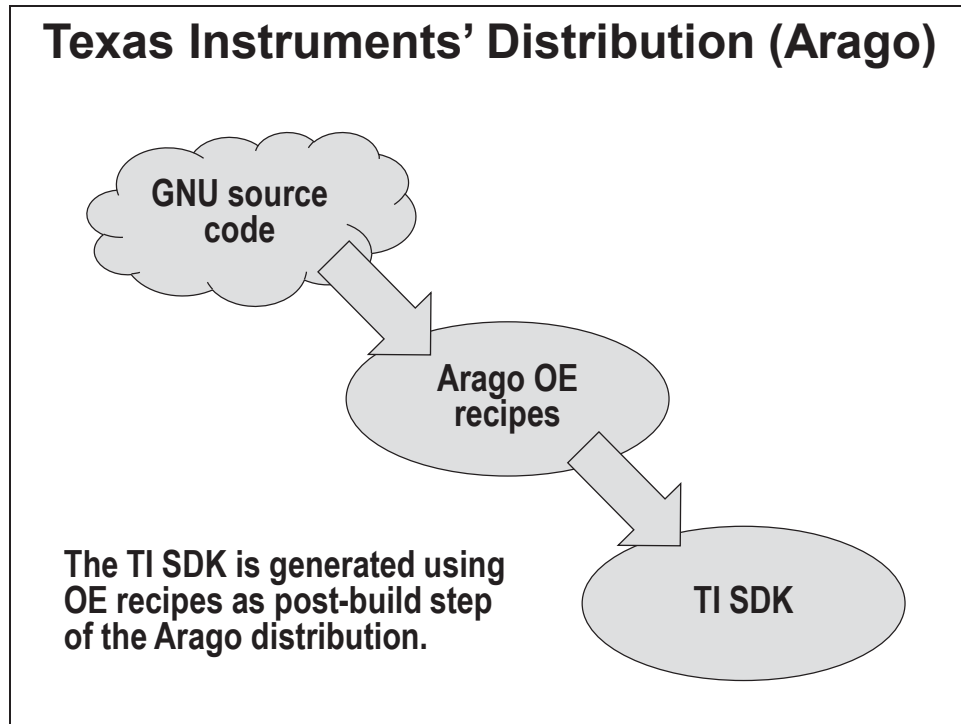
# Linux Application Development



Any product developer using Linux will eventually need to write one or more applications to drive their product.

Building an application that will execute on a Linux system requires a Gnu compiler chain (gcc) compliant toolchain. gcc-compliant toolchains embed the proper hooks as required by the ld loader program that is used to launch applications on a Linux system.

Certainly applications can be correctly built for a Linux system using a gcc compiler, but still care must be made to ensure that the application is built correctly. Firstly, since Linux relies heavily on shared libraries, it is important to make sure that the libraries that the application is compiled against are binary-compatible with the libraries that are installed on the target system. This is a particular concern when cross-compiling, as is commonly done for the development of embedded systems.



For this reason, TI provides a software development kit (sdk) for development on TI platforms. The sdk contains a pre-built Arago distribution, and contains the gcc compiler chain that was used to build the distribution as well as the shared libraries used so that applications built with this toolchain and library set should be compatible, although in some cases the compiler flags used to build applications may still cause compatibility issues if not set properly.

## Use the TI SDK if You Want...



### TI SDK

1. To take the Arago distribution “as is”
2. To use standard application build techniques (gcc, makefile)
3. A tested, supported Linux configuration

For users who wish to use the Arago distribution out-of-the-box without modification the sdk is generally sufficient to ensure compatibility. The main limitation of the sdk is that if one wishes to modify Arago into a custom distribution, this compatibility may be broken. However, many users do not wish to modify the Arago distribution because it provides a stable, tested platform for development. For those users, developing with the SDK is a good option and may be sufficient to meet their needs.

While less flexible than the approach of application building using Open Embedded recipes, this approach does have the advantage of simplicity, allowing the user to build their application using the familiar mechanism of a standard makefile-driven compiler without having to learn a new tool, namely bitbake.

## Modify the Arago OE Recipes if You Want...

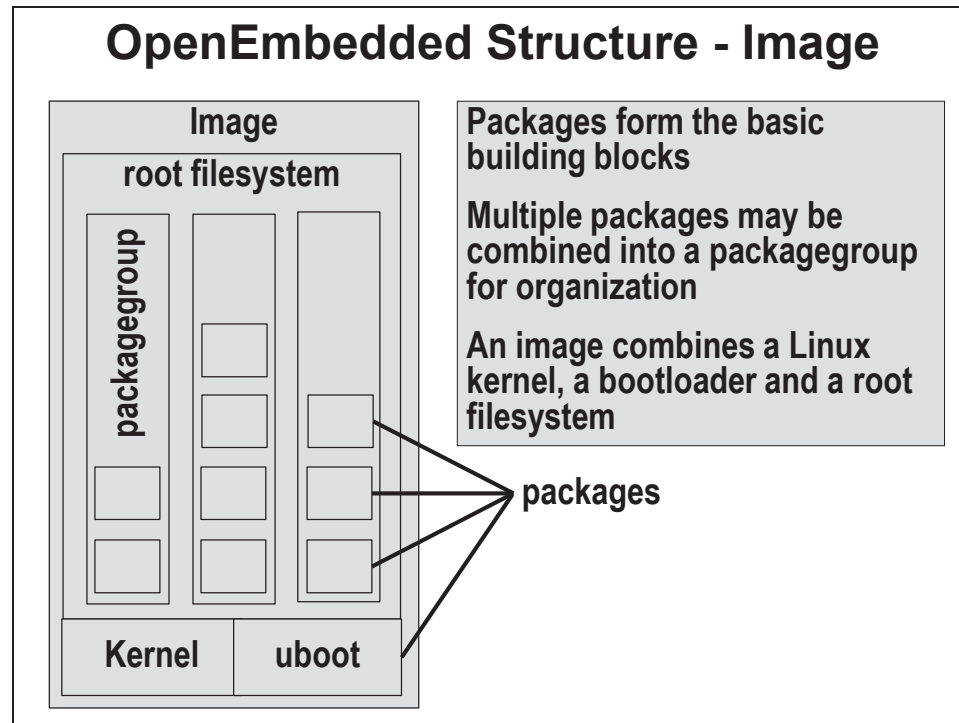
### Arago OE Recipes

1. To add GNU community software not supported in the Arago distribution
2. To remove unused components of the Arago distribution in order to use smaller and less expensive non-volatile memory
3. To more easily migrate to newer software releases and between hardware platforms

The recommended approach for application building, however, is to build the application within the Open Embedded build flow using recipe files. This approach has a couple of advantages. Firstly, since the same OE platform is used to build the application and the Linux distribution it runs on, compatibility is guaranteed, even if the user modifies the distribution to add new features or strip out unused components. In fact, the application becomes, effectively, just another component of the distribution that is built at the same time and using the same tools as the other components that form the distribution.

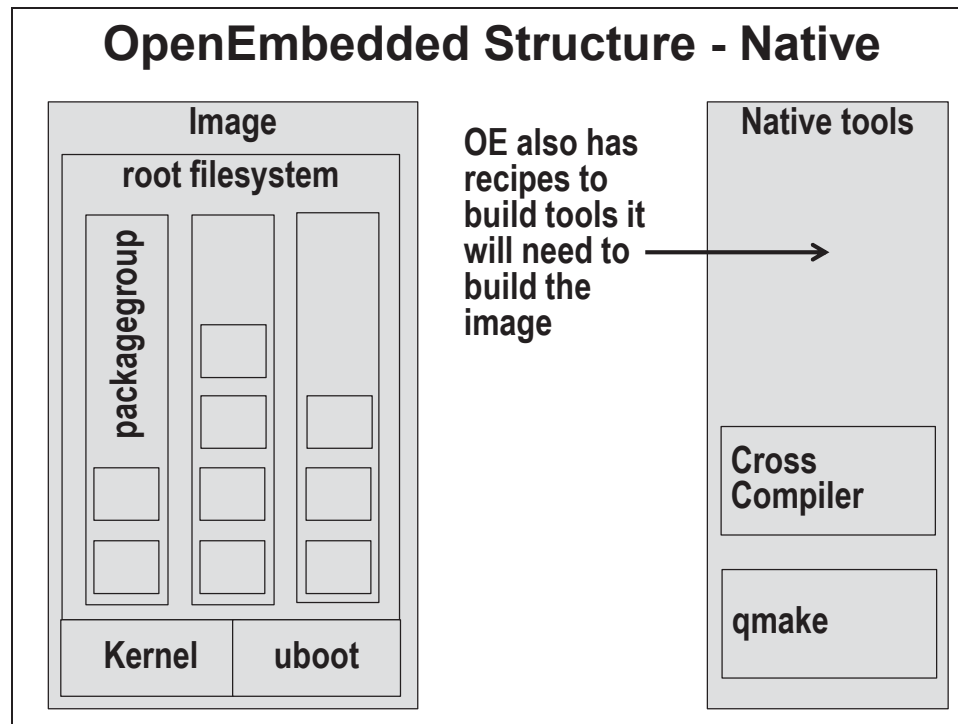
Even those users who do not plan to modify the Arago distribution may find it helpful to build their application within the OE bitbake recipe context for simpler migration to newer software releases and hardware platforms, whereas developing on a fixed SDK may require additional effort when porting the project forward to a newer SDK release.

## Bitbake Application Build



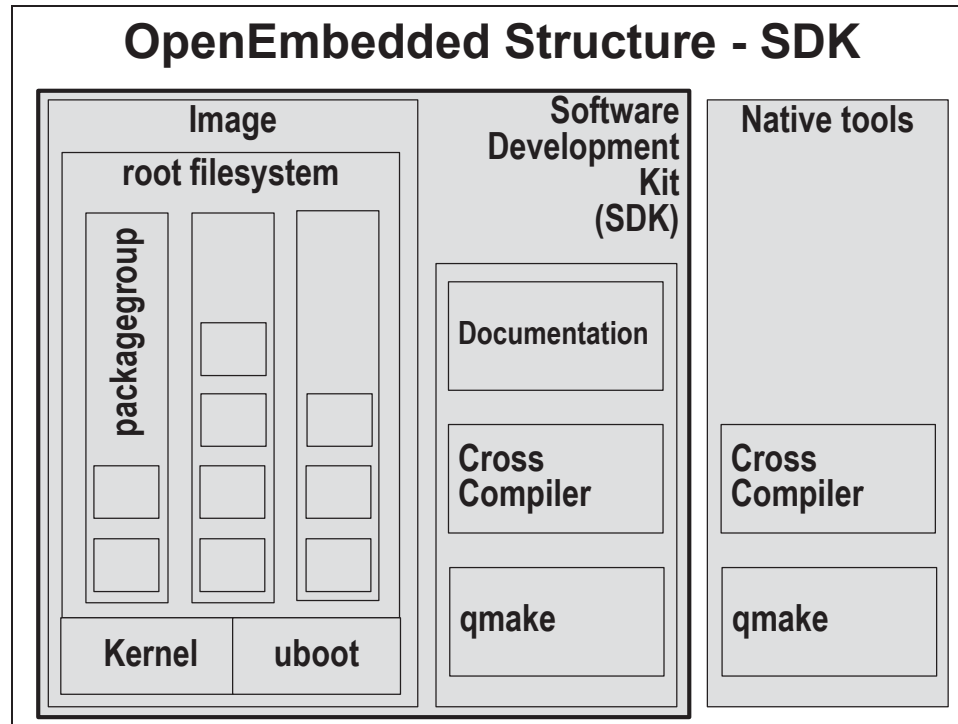
The basic building block of the bitbake tool is the package. Software components are grouped into packages, usually with a package containing the smallest grouping of a given software that would make sense to build on its own. These smaller package building blocks may then be grouped together into package groups of components which are often used in conjunction. Packages interact with each other in a number of ways, including the specification of dependencies so that a package which requires another package can inform the build tool of the requirement.

The ultimate goal of the tool is to build a Linux distribution, so there is a special package type called “image” that specifies an entire distribution image, including the root filesystem, the kernel and the bootloader. The largest of these three components is generally the filesystem, which is built from a number of packagegroups and individual packages. At the highest level, adding a software component to the distribution or removing it from the distribution is as simple as adding or removing the package from the image recipe.



Since openembedded builds distributions from source code, it requires build tools on the host computer. These tools are organized into a “native” package type, indicating that they are native to the host computer and not components to be executed on the target.

In order to guarantee the highest level of compatibility, Open Embedded actually has the ability to rebuild the gcc compiler chain used to compile and link the components of the distribution from gcc source code. Likewise it will rebuild other build tools, such as the qmake tool used to build qt applications.



Open Embedded even defines a package type (class) for building a software development kit for the Linux distribution that it generates. This software development kit contains all of the distribution components, the root filesystem, kernel and bootloader, and also contains all of the tools that were used to build those components and their associated libraries.

The Texas Instruments Software Development Kits that are released for the Arago distribution are built in this fashion using Open Embedded.

## Recipe Classes

Class functions are imported using “inherit” in a .bb recipe:  
`inherit native`

|                       |   |
|-----------------------|---|
| <b>base</b>           | the default package build recipe          |
| <b>qt4e</b>           | build a qt/e project (non-X11)            |
| <b>image</b>          | build a root filesystem image             |
| <b>native</b>         | build a host exec to use during build     |
| <b>nativesdk</b>      | build a native executable for sdk         |
| <b>crosssdk</b>       | build a native cross compiler for sdk     |
| <b>cross-canadian</b> | build a non-native cross compiler for sdk |

`$ find <oedir>/sources -name "*.bbclass"`

A distribution image is comprised of a number of different components, and not all of these components may be built using the same build flow. For this reason, bitbake packages are capable of inheriting various classes which define how their components are built. Some common classes are listed above. The base class can be used to build most application code. qt projects require a different build flow, so they have their own class definition. Classes can also be used to specify different build flows depending on whether the application is to be built for the target (i.e. the base class) the host computer (the native class) or even if the host computer is to build a cross compiler to run on another host type (cross-canadian)



## Base Class Tasks

**\$ more sources/oe-core/meta/classes/base.bbclass**

|                         |   |
|-------------------------|---|
| <b>do_fetch</b>         | Fetch source using url (http://, ftp://, file://, git://, etc) into \${WORKDIR}                       |
| <b>do_unpack</b>        | Unpack archives (.tar.gz, .tar.bz, etc) from \${WORKDIR} into source directory, \${S}                 |
| <b>do_patch</b>         | Apply patches to source code in \${S}   |
| <b>do_configure</b>     | Configure source code package in \${S}  |
| <b>do_compile</b>       | Compile source code and place into \${B}  |
| <b>do_install</b>       | Populate the package install directory, \${D}   |
| <b>do_package</b>       | Split contents of install directory among various packages (-dev, -dbg, etc) and place in \${PKGDEST} |
| <b>do_package_write</b> | Archive package directories using .ipk, .deb or .rpm and place into \${DEPLOY_DIR_<IPK,DEB,RPM>}      |

**Base implementations of “do\_compile” and “do\_install” assume a makefile has been provided with the source code.**

The base bitbake class defines a set of tasks that are used to build a base package. Since the other class types inherit from the base class, most class types follow this same basic build structure, although many classes extend the basic list with additional tasks, and in some cases plug one or more of the base tasks with an empty stub, effectively removing it.

As these build flow tasks execute, they move their respective products along a chain of directories. This is useful for two reasons. Firstly, by moving the products of each successive task forward, the history of the build procedure results is maintained. Once a given task is executed, it does not need to be repeated unless a change is made to the package recipe that forces a re-execution of the task.

Secondly, maintaining the results of each task step in separate directories makes the debugging process simpler. If a recipe developer does not find their expected output upon building a new recipe, he or she can trace through the various directories in the build flow to determine at what point in the build process produced unexpected results.

## Helloworld Bitbake Recipe File

```
LICENSE = "PD"
LIC_FILES_CHECKSUM="file://${S}/LICENSE;md5=d41d8cd98f00
b204e9800998ecf8427e"
SRC_URI = "file://main.c file://LICENSE "

S = "${WORKDIR}"
do_compile () {
    ${CC} ${S}/*.c ${CFLAGS} ${LDFLAGS} -o hello
}

do_install () {
    install -d ${D}${bindir}
    install -m 0755 ${S}/hello ${D}${bindir}/hello
}
```

**This example overrides the base class implementations of “do\_compile” and “do\_install”**

This simple recipe file demonstrates a package recipe which will build a hello world application from source code and package the resultant executable into an installable package. The example also demonstrates the use of task override.

The base class task of bitbake defines task functions for each task in the build flow that was shown on the previous slide. Most of the tasks defined meet the needs of this package; however, the base class expects the source code for the application to contain a makefile script to handle compiling and installing the application, and the implementation of the base tasks simply calls that makefile in order to complete these steps.

In this example we wish to build the application directory instead of invoking a makefile. Thus the recipe contains overrides for the “do\_compile” and “do\_install” tasks. Note the usage of a number of bitbake-defined variables such as “\${CC},” the variable which invokes the bitbake c compiler corresponding to the target that is being built. It is important for recipe developers to use these built-in variables whenever possible to guarantee portability of the recipe both across host build systems and across deployment targets.

## Important Recipe Directories

### **`${WORKDIR}`**

The working directory. Bitbake copies the source to this directory, unpacks it and then configures and builds it

### **`${S}`**

The source directory. Source code is often downloaded as an archive (.tar.gz) and then de-archived into this directory

### **`${B}`**

The build directory, where bitbake places executable after `do_compile`. By default it equals `${S}`

### **`${D}`**

The destination directory. Any files copied into this directory are published as part of the package.

This slide shows a list of the build directories commonly invoked in bitbake recipes.

Note that the base class tasks use these directories, invoked as variables, as the source and destination directories of their individual operations. Thus, if a given task in the chain is overridden, it is important to use these directories as the source and destination for the task to assure that it inserts correctly into the chain of build commands.

Furthermore, the bitbake build system provides an organizational structure that organizes various host tools as well as the package builds for different targets. This organizational structure is reflected in the directory variables above, and should be adhered to in all recipes in order to maintain a parallel organization of the build tree.

## Important Recipe Build Variables

**`${CC}`**

The calling function (including path) for the system C compiler

**`${CFLAGS}`**

The default C compiler flags

**`${CXXFLAGS}`**

The default C++ flags

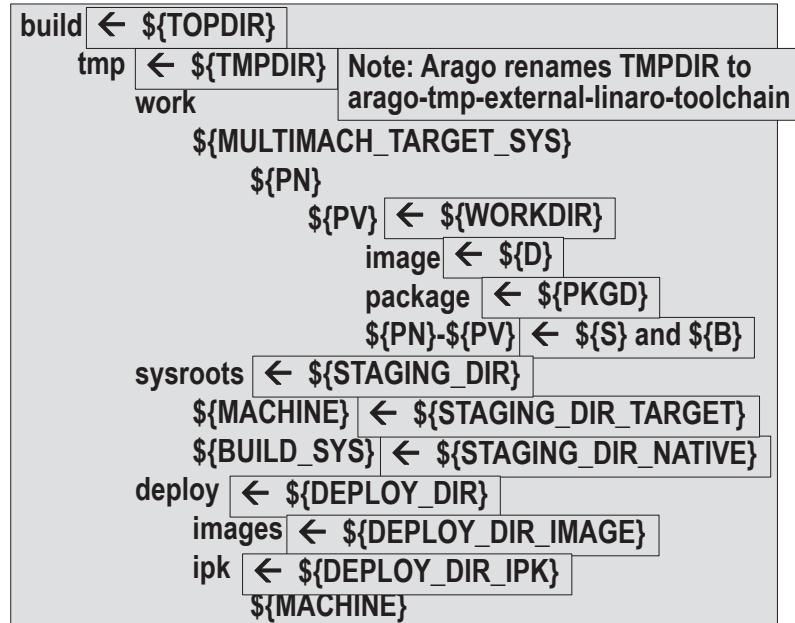
**`${LDFLAGS}`**

The default linker flags

This slide shows variables commonly used to implement the “do\_compile” build task of bitbake recipes. By using the system variables to provide flags to the c compiler and linker, the recipe developer guarantees that the same compiler and linker flags are being used to build this package code as are being used to build the other packages in the system. This is important for compatibility of the final product, which is an entire Linux-based system.

Utilization of the compiler calling function and various compiler and linker flags also provides portability of the recipe across platforms. One of the most important features of bitbake and the open embedded build system is the ability of a single recipe to build code for a wide array of platforms, allowing reuse of recipes across the community. This also provides a benefit to customers who wish to maintain their application code across multiple platforms or would simply like to reduce the effort involved in porting their application forward to newer Linux distribution releases as well as new hardware platforms.

## Open Embedded Build Directories



This slide shows the directory structure that is used by bitbake for building a package. It shows the directories that are commonly seen in recipe tasks, `${WORKDIR}`, “`${D}`,” “`${S}`” and “`${B}`.” It also shows some of the directory variables that are less commonly used in recipes but may come up during the recipe debugging process.

The variables shown in the tree are the variables used to define the directory and the variables shown next to the arrows are the variables used to reference the directory. So, for instance, the `${WORKDIR}` variable evaluates to “`build/tmp/work/${MULTIMACH_TARGET_SYS}/${PN}/${PV}`” where `${PN}` is the package name and `${PV}` is the package version.

This slide is presented as a reference and contains much more information than is generally required to write a bitbake recipe, so do not be overwhelmed.

## Using opkg to Install New Packages

**To see the commands the opkg package manager supports**

```
host$ opkg
```

**To install an opkg-compliant package**

```
host$ opkg install <packagename>.ipkg
```

**To download a package but not install it**

```
host$ opkg download <packagename>
```

**To find out which packages provides a given file**

```
host$ opkg whatprovides <filename>
```

**To list which packages are available**

```
host$ opkg list
```

**To update the list of available packages and “whatprovides” search**

```
host$ opkg update
```

Packages archive their contents using one of three formats: .ipk, .deb or .rpm

All three of these formats are from well-established package managers within the Linux community. The Arago distribution uses the OPKG package manager, which installs .ipk archives.

Installing an .ipk archive is as simple as executing the command “opkg install” as shown above. Some of the other features of the opkg package manager are shown above.

## Lab 3: Open Embedded Hello World

### Lab 03 Key Concepts

#### 1. Inheritance

We want to tell bitbake to use the “native” class to build an i686 app instead of a Cortex-A8 app

#### 2. Source directory override

`${S} = ${WORKDIR}`

#### 3. Task overrides

The default “do\_compile” and “do\_install” tasks utilize a makefile. Instead of implementing a makefile, we will override these tasks to directly invoke the C compiler and linker.

#### 4. Output in sysroot

The “native” class will output our executable to the i686 sysroot

(Page intentionally left blank)



# Module 04: Application Debug

---

## Introduction

Code Composer Studio version 6 is the current iteration of Texas Instrument's integrated development environment. This IDE can be used to program any of TI's processors from the low-cost, ultra-low-power MSP430 all the way up to the ultra-performance Arm Cortex-A8 devices such as the AM335x.

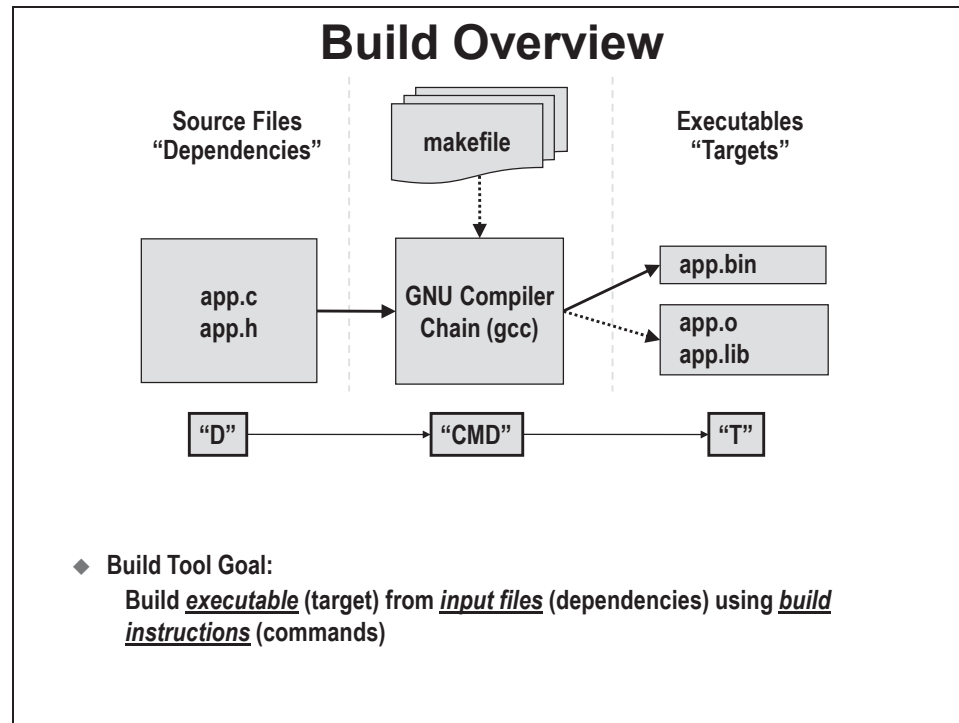
CCSv5 is based on eclipse, an open-source IDE. This provides many advantages. Firstly, customers who are familiar with other eclipse-based IDEs will have less learning curve, and those who are not are likely to run into eclipse-based IDEs in the future. Secondly, there is a large community of eclipse plugins that can be used with CCSv6. Also, eclipse runs on both Windows and Linux host machines.

This module will introduce the basic features of CCSv6 and provide details on setting up a debugging environment.

## Module Topics

|   |             |
|---|-------------|
| <b>Module 04: Application Debug.....</b>            | <b>4-1</b>  |
| <i>Module Topics.....</i>                           | <i>4-2</i>  |
| <i>Intro to Makefiles.....</i>                      | <i>4-3</i>  |
| <i>Code Composer Studio Overview .....</i>          | <i>4-11</i> |
| <i>CCS-Compatible Project for OpenEmbedded.....</i> | <i>4-20</i> |
| <i>Setting up a CCSv6 GDB Session .....</i>         | <i>4-28</i> |
| <i>Lab 4: Application Build and Debug .....</i>     | <i>4-33</i> |

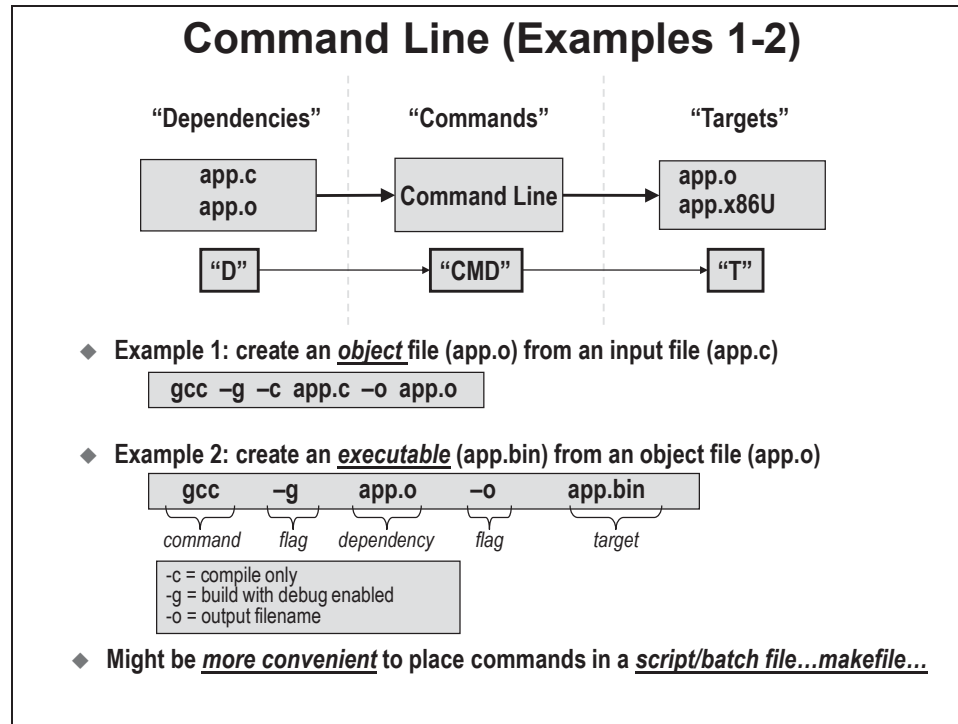
# Intro to Makefiles



Building executable programs follows a standard flow. Source files, such as “.c” source files and “.h” header files in the C programming language are written by the program developer. These source files are compiled and linked by build tools into an executable binary (“.bin”) file. Or, in some cases, these source files are instead built into intermediate object files (“.o”), which are compiled but not linked, or they are linked into a library instead of an executable. With both the library and object files, these intermediate products become source files in a second pass of the build tool, the ultimate objective being to produce an executable application.

While this procedure may be completed from the command line without a scripting utility, it is laborious, especially when source files are repeatedly modified during development and need to be rebuilt for testing. Thus most developers utilize some type of build scripting utility to drive the process. Here we show the “makefile” scripting utility, which was developed specifically to script the building of applications.

Below the source files, GNU compiler chain and binary files above we see the labels “D” for Dependency, “CMD” for command and “T” for target. These labels will be discussed further in regards to their significance in writing makefile rules.



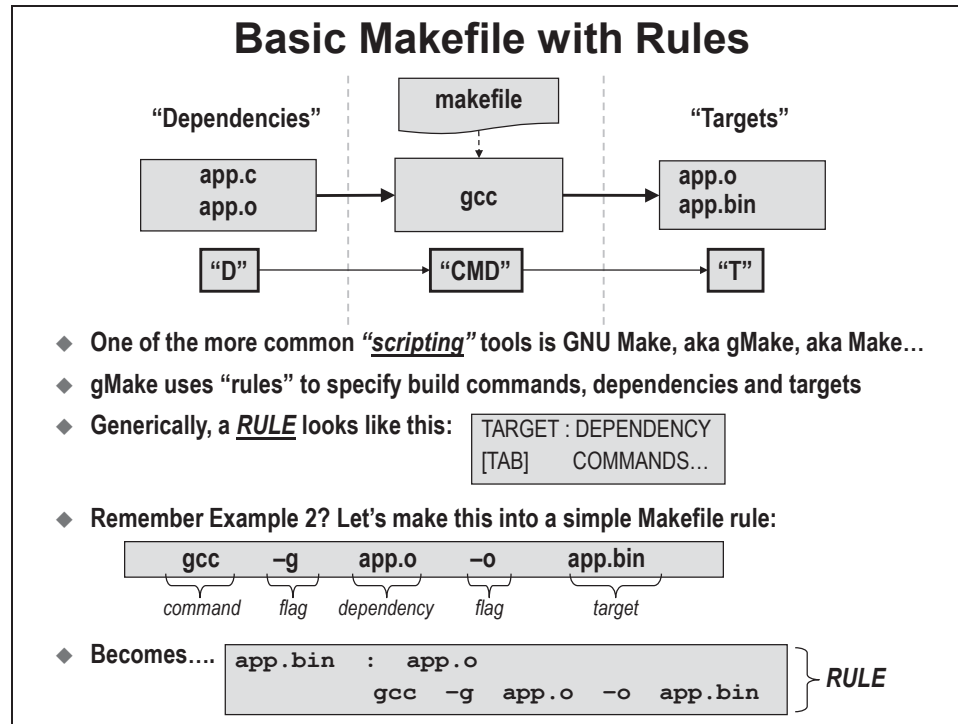
Before exploring makefile scripting, it is useful to review the procedure for building applications from the terminal command line. For Linux, the application used to compile, assemble and link programs is called “gcc,” which stands for “GNU compiler chain.”

In its simplest format, gcc can be invoked with a list of source files as parameters, and it will compile, assemble and link them as necessary to form an executable, which by default it names “a.out”

The examples above show a few other common flags which are used with gcc. In Examples 1 and 2, the “-o” flag is used to specify the name of the output file so that in example 1 the file created by gcc is renamed “app.o” and in the second example it is names “app.bin”

Additionally both examples use the “-g” flag to embed debugging information. Finally, example 1 uses the “-c” flag. This flag instructs the gcc tool to only compile the provided source files and not attempt to then link the result into an executable, which would be the default behavior. Thus in the examples shown, the command line of “Example 1” will combile app.c into app.o, and the command line of Example 2 will then link app.o into app.bin

The usefulness of separating the compile and link steps will become more apparent as makefile rules are discussed. Using the timestamp capability of makefiles, separating into two steps will provide shorter build times in many cases by not forcing the tool to recompile files that have not changed since the previous build.



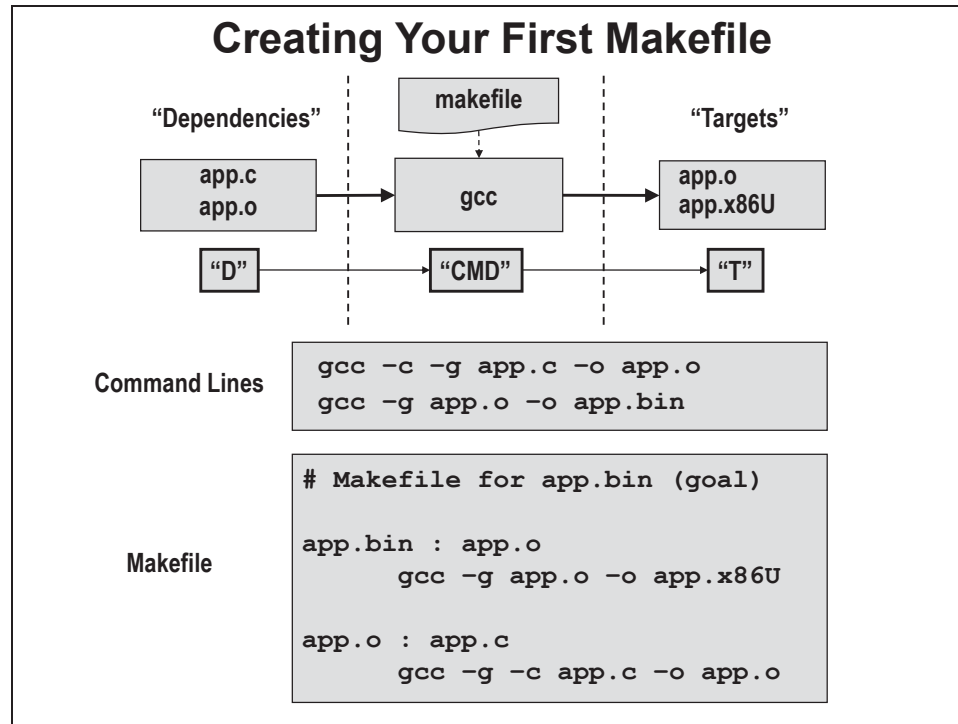
The basic units of makefile scripts are build rules. This slide shows how to convert the command line shown as “Example 2” on the previous slide into a makefile rule.

Generically rules consist of a header declaration that defines the target and dependency followed by a list of commands that build the declared target. The target is separated from the dependency list by a colon. The dependency list is a set of file that the target is dependent upon. When the target is set to be rebuilt, the makefile utility will check the timestamps of every file in the dependency list against the timestamp of the target. If any file in the dependency list has changed since the last build of the target, makefile will execute the commands from the command list. If the timestamp on the target is instead newer than the timestamps on every file in the dependency list, then the target is up to date and the rule is not executed.

Note that whitespace characters are significant in makefile scripts. The files in the dependency list are separated by one or more space characters. The target:dependency header is separated from the command list by a single carriage return, and the commands in the command list must be preceded by a tab character (not five presses of the space bar!)

The example at the bottom shows a simple implementation of a makefile rule to build “app.bin” The target is the file that the rule will build, app.bin. This is separated from the dependency list by a colon. In this case there is a single file in the dependency list, which is app.o, because if app.o changes, it needs to be re-linked into a new app.bin

Finally, the command to convert app.o into app.bin is entered under the header. In this simple example it is exactly the text that one would type at the command line in order to link app.o into app.bin, although in real makefiles it might include variables or makefile functions as we will see.



By adding a second rule, a working makefile is created that will create `app.bin` from the `app.c` source file. Here there is a rule to compile `app.c` into `app.o` and a rule to link `app.o` into `app.bin`. The makefile utility will parse these rules and, because `app.o` is the target of one rule and the dependency of another rule, will chain the rules together into a dependency tree. This means that if `app.c` is modified and the user invokes the makefile utility with `"app.bin"` as the target, the utility will first execute the rule to compile `app.c` into a new `app.o` and then will invoke the rule to link `app.o` into `app.bin`. The utility is capable of linking as many rules together in this manner as is necessary, and the building of this dependency tree does not require any specific ordering of the rules that form the tree, so that even if the order of these two rules was reversed, the utility would still chain the two rules together to build `app.bin` from `app.c`.

One might ask if it would not have been simpler in this example to build a single rule to compile and link `app.c` into `app.bin` in one step. In this example that would probably have been a better approach because there is only a single `".c"` source file; however, in most real-world examples there are multiple `".c"` source files to be compiled and eventually linked together into the final executable, and in that case, separating the compile and link steps is a more efficient approach because it does not force unchanged files to be recompiled. (Recall that the makefile utility will only execute a rule if the timestamp of the target is older than the timestamp of one or more files in the dependency list.)

## User-Defined Variables & Include Files



- ◆ User-defined variables simplify your makefile and make it more readable.
- ◆ Include files can contain, for example, path statements for build tools. We use this method to place absolute paths into one file.
- ◆ If “include path.mak” is used, the “-” tells gMake to keep going if errors exist.
- ◆ Examples:

### makefile

```
include path.mak
CC := $(CC_DIR) /gcc
CFLAGS := -g
LINK_FLAGS := -o

app.bin : app.o
    $(CC) $(LINK_FLAGS) app.o -o app.bin
```

### path.mak

```
CC_DIR := /usr/bin
...
# other paths go here...
```

While the previous example is a working makefile, most makefile authors will make use of variables. This example shows a common usage of storing compiler and linker flags in a variable and then referencing that variable throughout the makefile. It also shows how an external file may be included into the makefile using the “include” command. In this example the included file has a “.mak” extension, which is a common style, but is not required.

Variables are declared with either the “:=” (canonical equals) or the “=” (equals) notation. The difference between these two assignments occurs when there is a dereferenced variable on the right-hand side of the assignment. The canonical equals (“:=”) immediately dereferences all variables on the right-hand side and uses their current value at that point in the script to set the variable. This is referred to as immediate assignment.

The “=” (equals) notation maintains any variable reference on the right-hand side of the assignment as a variable so that each time that the left-hand-side variable is dereferenced, all variables on the right-hand side are dereferenced. This is referred to as deferred assignment.

## Using Built-in Variables



◆ **Simplify your makefile** by using these built-in gMake variables:

- `$@` = Target
- `$^` = All Dependencies
- `$<` = 1<sup>st</sup> Dependency Only

◆ **Scope** of variables used is the current rule only.

◆ **Example:**

*Original makefile...*

```
app.bin: app.o
    gcc -g app.o -o app.bin
```

*Becomes...*

```
app.bin: app.o
    gcc -g $^ -o $@
```

There are a second type of variables, which are built-in variables. Unlike the user-defined variables shown on the previous slide, which have a global scope, the scope of these built-in variables is the rule that they are referenced within. For instance, the “`$@`” variable is replaced with the target of the rule it is located in.

One might question whether it makes sense to use these variables to replace the target and dependencies of a simple rule as in the example above, since the only value of the built-in variables, in this example, is to save the author a few keystrokes and it reduces the readability of the rule.

In fact, while the above example is a working makefile rule, this usage is not the real purpose of these built-in variables. Their real purpose is in writing rule templates, as will be shown on the next slide.



## Templated Rules Using “%”

- ◆ Using pattern matching (or pattern substitution) can help simplify your makefile and help you remove explicit arguments. For example:

### Original Makefile

```
app.bin : app.o main.o
    $(CC) $(CFLAGS) $^ -o $@

app.o : app.c app.h
    $(CC) $(CFLAGS) -c $< -o $@

main.o : main.c main.h
    $(CC) $(CFLAGS) -c $< -o $@
```

### Makefile Using Pattern Matching

```
app.bin : app.o main.o
    $(CC) $(CFLAGS) $^ -o $@

%.o : %.c %.h
    $(CC) $(CFLAGS) -c $< -o $@
```

- ◆ The .bin rule depends on the .o files being built – that’s what kicks off the .o rules

A very powerful capability of makefile scripts is rule templating. Rule templates use pattern matching to specify the build rules for multiple targets. For instance, in the above example the rules for building “app.o” and “main.o” use exactly identical commands. Because this is the case, those two separate rules can be replaced by a single rule template, which specified the build instructions for building any “.o” object file from a “.c” c-source file of the same name.

It should be noted that without the built-in variables, it would not be possible to write the command for the “app.o” and “main.o” targets to be character-by-character identical. This is why the comment was previously made that the real utility of built-in variables is in creating rule templates.

Rule templates use the percent symbol (“%”) to perform pattern matching in the target and substitution in the dependency list. Any rule template must contain a “%” in the target portion of the declaration. The makefile utility will attempt to match build targets to this pattern, so that, because the “app.bin” target depends on “app.o,” whenever “app.bin” is built, the makefile utility will attempt to match “app.o” to “%.o” and will succeed with the relation “%=app”

Once a match to the template is made, the “%” is substituted anywhere that it appears in the dependency list. In this example “%.c” will be expanded into “app.c” and “%.h” will be expanded into “app.h” The “%” may appear as many times as required in the dependency list, including zero times.

## Makefile Functions

**Makefile functions are called with a command and its arguments placed inside parenthesis and preceded by the dollar sign**

```
$(command args)
```

**The return value of the function is placed in-line to the makefile.**

```
C_FILES := $(wildcard *.c)
```

**Makefile functions may also be nested**

```
# equivalent to:  
# C_FILES := $(wildcard *.c)  
# STRIPPED := $(notdir $(C_FILES))  
  
STRIPPED:= $(notdir $(wildcard *.c))
```

Another powerful capability of makefiles is the calling of built-in functions. The user may find that many of these functions can be replaced by similar functions and expansions within the command lines of given rules, which are then expanded within the shell; however, it is recommended to use the makefile functions instead. This will add portability as the makefile functions will be implemented by the makefile utility in whatever manner is appropriate for the system the utility is executing on. Also, makefile functions may be called anywhere within the makefile script, whereas shell functions and expansions may only be used within the command list of the script rules.

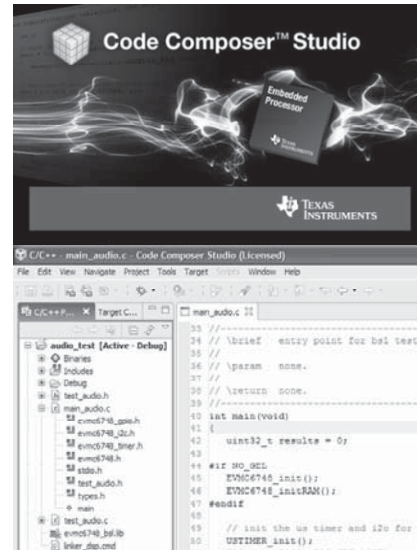
Makefile functions are invoked with a dollar sign and parenthesis. Both the function and any arguments to the function are specified within the parenthesis, and the return value of the function is inserted as text in-line to the script at the location that the function is called.

Makefile functions may even be nested. In the example above, the return value of the “wildcard” function is used as the parameter passed into the “notdir” function. The “wildcard” function will expand into a space-separated array of files in the current directory which end in the extension “.c” and the “notdir” function will strip the path from each file in the array.

# Code Composer Studio Overview

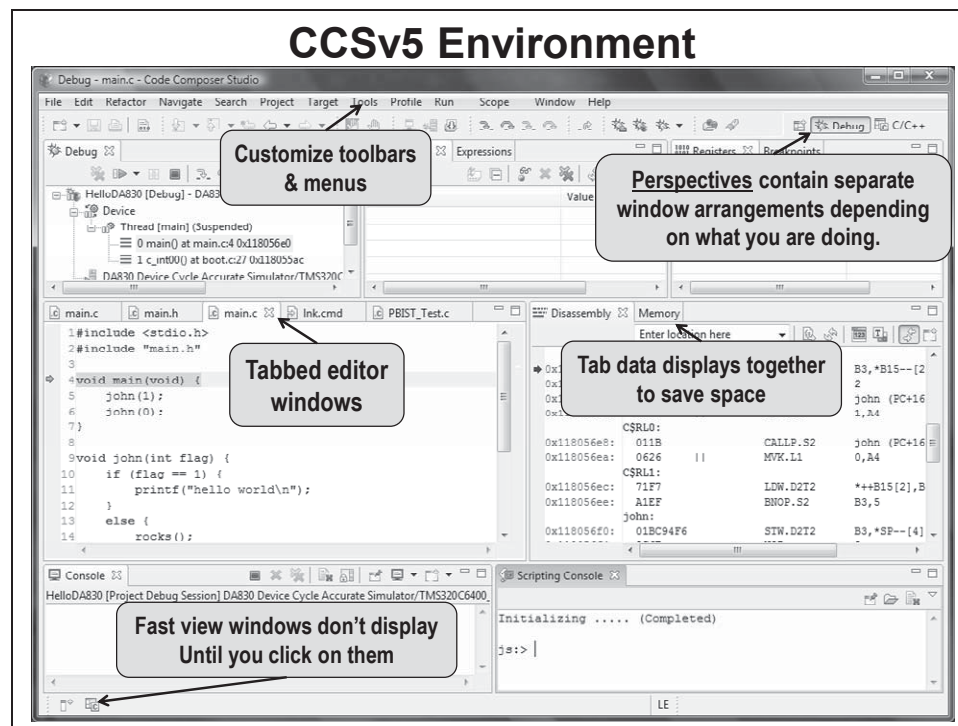
## What is Code Composer Studio?

- ◆ Integrated development environment for TI embedded processors
  - ◆ Includes debugger, compiler, editor, simulator, OS...
- ◆ CCSv5 is based on “off the shelf” Eclipse
  - ◆ TI contributes changes directly to the open source community
  - ◆ Drop in Eclipse plug-ins from other vendors or take TI tools and drop them into an existing Eclipse environment
  - ◆ CCSv5 runs on a Windows or Linux host
- ◆ Integrate additional tools
  - ◆ OS application development tools (Linux, Android...)
  - ◆ Code analysis, source control...
- ◆ Low cost!



Code Composer Studio (CCS) is Texas Instruments’ recommended tool for building and debugging Linux applications on TI ARM-based products. In fact, CCS is TI’s recommended tool for software development on all of TI’s processors.

CCS is based on the elipse integrated development environment. Eclipse is an open-source IDE that will run on a Windows or Linux host. Because CCS is based on “off the shelf” elcipse, users can extend the platform by install eclipse plugins from the eclipse community.

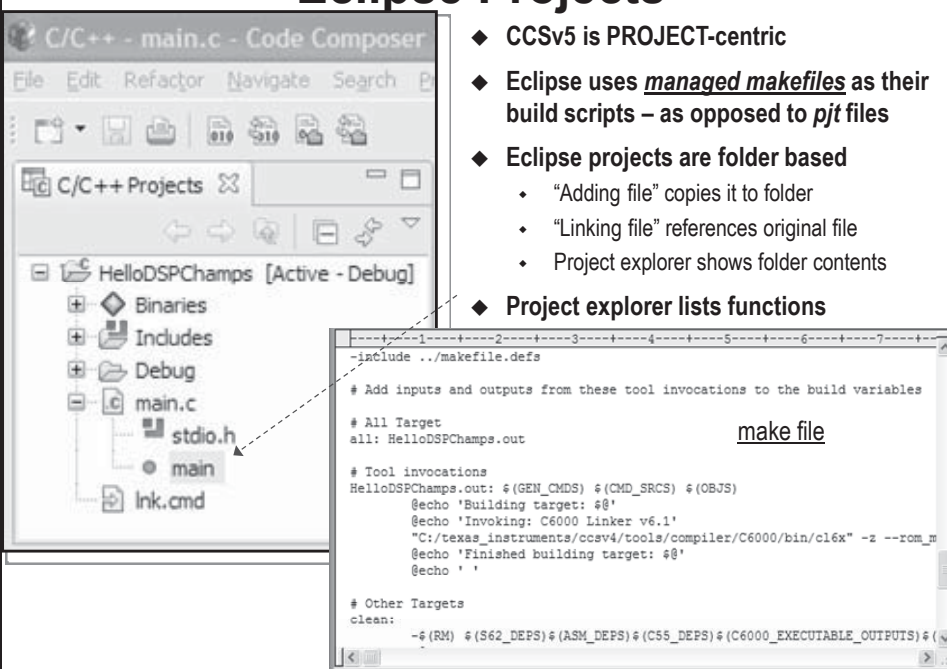


Eclipse has a number of features to aid code development and debugging. The tabbed editor and display windows help make efficient use of the display, and the ability to customize toolbars and menus gives “power” users the ability to reshape the IDE to meet their needs.

Perspectives are an excellent time-saving tool. Users can save a set of eclipse windows for different tasks. For instance, CCS is delivered with “debug” and “C/C++” perspectives out of the box. The debug perspective provides windows useful for debugging, as shown on the screen capture. The “C/C++” perspective is more useful for code development when the debugger is not used and removes the debugging-centric windows in order to add the project explorer window and expand the side of the editor window.

Utilizing these two perspectives, a user can switch between these sets with a single button instead of having to manually close unused windows and then open new windows from hot keys or menu pull-downs.

## Eclipse Projects



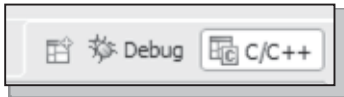
- ◆ CCSv5 is PROJECT-centric
- ◆ Eclipse uses managed makefiles as their build scripts – as opposed to *pjt* files
- ◆ Eclipse projects are folder based
  - “Adding file” copies it to folder
  - “Linking file” references original file
  - Project explorer shows folder contents
- ◆ Project explorer lists functions

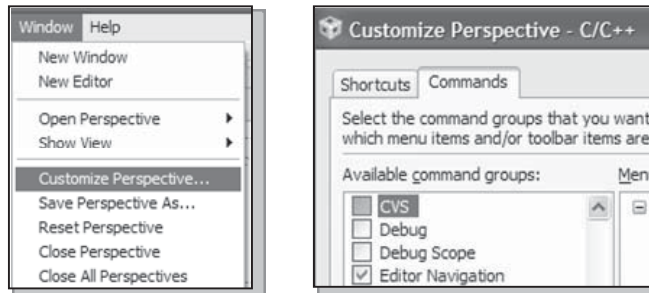
Eclipse manages application projects using a project folder on the host filesystem and a makefile script for building. When a new project is created within eclipse, the IDE creates a new folder to hold the files associated with the project. Most of these are source files, although there are also hidden metadata files. Files may be brought into the project build in one of two fashions. If the file is “added” eclipse will copy the file into the project folder. If the file is “linked,” eclipse will instead use a link to the original source file. Modifying an “added” file will not modify the original, but modifying a “linked” file will. Which of the two methods is appropriate depends on what the user is trying to accomplish.

The default behavior of eclipse is to use a managed makefile for the project build. A managed makefile is created and updated by the eclipse IDE, and, though the developer may open the file to examine the contents for debugging purposes, should not ever be modified directly by the user. Users who prefer the added level of control of writing their own makefile may do so, but it should be noted that the makefile is either exclusively controlled by the eclipse ide or exclusively controlled by the user; it is not possible to toggle back and forth.

## Perspectives

- ◆ **Perspectives** – a set of windows, views and menus that correspond to a specific set of tasks
- ◆ Two **default perspectives** are provided with CCSv5:
 

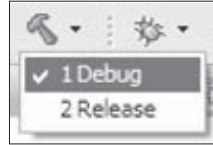
|  |  |   |
|--|--|---|
| <b>Debug</b> <ul style="list-style-type: none"> <li>• Debug Views</li> <li>• Watch/Memory</li> <li>• Graphs, etc.</li> </ul> |  | <b>C/C++</b> <ul style="list-style-type: none"> <li>• Code Dev't Views</li> <li>• Project Contents</li> <li>• Editor</li> </ul> |
|--|--|---|
- ◆ Users can **customize perspectives** and save them:



The perspective capability described previously can be customized in two ways. Firstly, the user can change the windows displayed by an existing perspective using the “customize perspective...” function. Secondly, users can create any number of new perspectives by utilizing the “save perspective as...” option.

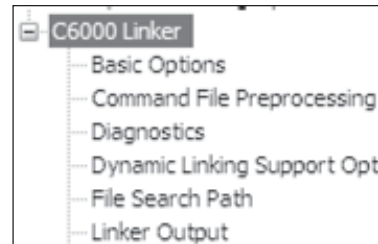
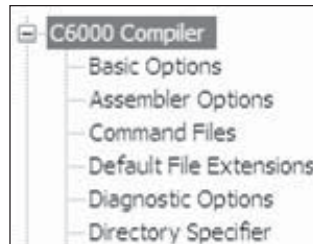
## Two Default Build Configurations

- ◆ **Build Configuration** – a set of build options for the compiler and linker (e.g. optimization levels, include DIRs, debug symbols, etc.)
- ◆ CCSv5 comes standard with two DEFAULT build configs: Debug & Release:



User can create their own config if desired

- ◆ User can modify compiler/linker options via "Build Options":



Many more details on compiler options in a later chapter...

18

When eclipse is managing the project makefile, the ide writes makefile rules for the user based on the project type and the source files added into the project. When the ide writes a rule to compile or link, it must specify flags to the compiler or linker used. There are a default set of flags that the ide uses, but many users may wish to change these flags. This is accomplished via build configurations.

The options that the user selects for a build configuration translate directly into the compiler and linker flags used to build the project. Additionally, the user can specify multiple build configurations to support different requirements. CCS is delivered with two build configurations: Debug and Release. The debug configuration specifies the compiler flags required to embed symbols into the executable that the user can use for debugging the application. The release configuration removes the debugging symbols and their associated cycle and memory overhead, and increases the optimization of the compiler for improved performance.

## Debugging & Debugger's

### ◆ User Mode Debugging

- When debugging user mode programs, you often only want to debug – hence stop – one thread or program.
- GDB (GNU Debugger) works well for this. (GDB discussed on next slide)
- Connection is usually Ethernet or serial-port

### ◆ Kernel Mode Debug

- Debugging kernel code requires complete system access.
- You need KGDB (Ethernet) or scan-based (JTAG) debuggers for this.

### ◆ A debugger lets you

- Pause a program
- Examine and change variables
- Step through code

### ◆ Code Composer Studio (CCSv6)

- Latest version of TI's graphical software debugger (i.e. IDE)
- IDE – integrated development environment: which combines editing, building and debugging into a single tool
- Built on Eclipse platform; can install on top of standard Eclipse
- Allows debugging via GDB (Ethernet/serial) or JTAG (scan-based)
- Free license for GDB debugging

Applications may be debugged using either software or JTAG-based debugging. Hardware-based debugging using a JTAG pod is generally only useful for debugging the Linux kernel or bootloader programs such as u-boot because these programs use physical addresses and the JTAG as a standard only works with physical addresses.

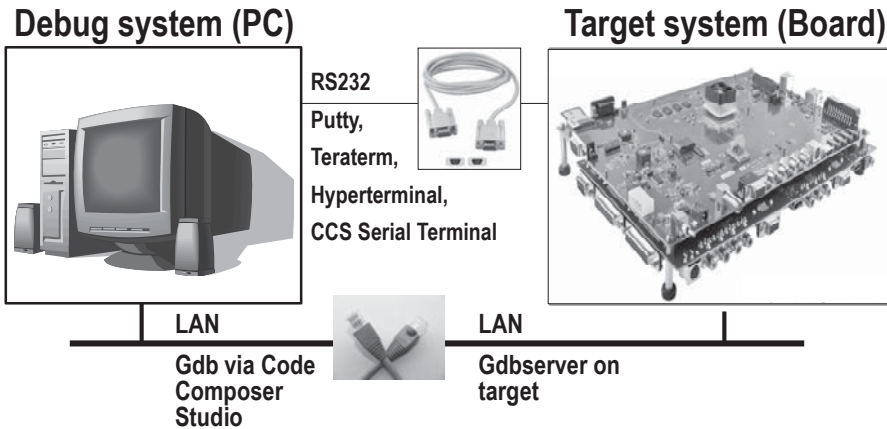
Software based Linux debugging uses the GNU debugger (gdb) and is most useful when debugging user applications because gdb is capable of doing the virtual-to-physical address translation that JTAG is not. One of the primary advantages of JTAG is that it maintains debugging capability even if the Linux kernel crashes, allowing “post mortem” debugging. The gnu debugger executes as an application running in the context of the Linux operating system, so if the Linux kernel crashes, the gnu debugger will stop functioning. This is not typically a concern when debugging user applications because, while it is certainly possible to crash the application, it should be not possible for an application to crash the Linux kernel thanks to the memory protection afforded by the Memory Management Unit.

Of course, a very popular point in favor of the gnu debugger is that, since it is open-source software, it is free to use, whereas JTAG-based debugging requires a JTAG pod, which has an associated cost, and a JTAG-capable license of CCS, which has its associated cost as well.



## Linux application debug setup

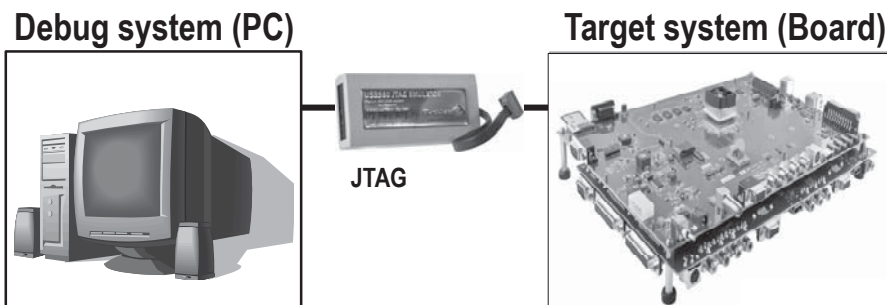
- Physically the connections of a typical linux application debug system are shown below.



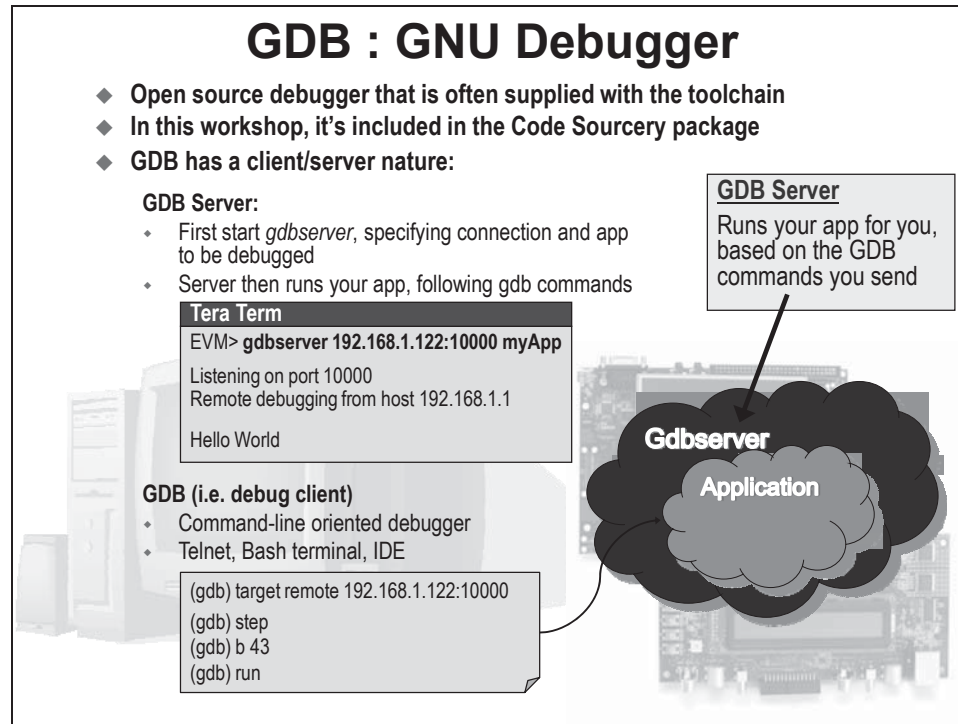
The only physical connection required to debug a remote target using gdb is an Ethernet cable. Often if there is a serial connection available, developers will also connect a serial terminal for added visibility into the system, although this is not required.

## Linux kernel and driver debug setup

- Physically the connections of a typical uboot, linux kernel or driver debug system are shown below.



For kernel debugging, the only connection required is via a JTAG pod, which connects to the board via a standard header and into the PC using either USB, Ethernet or PCI.



Code Composer Studio will initiate gdb debugging sessions from the push of a button; however, it is useful to understand the underlying mechanism both for setting up CCS and in case the user ever wanted to debug using gdb outside of the scope of CCS.

When debugging a native application, for instance debugging an x86 program on an x86 host, one may simply invoke the GNU debugger with the application as an argument. However, in our case we wish to use our x86 host as the debugging platform but debug a program that is running on the Sitara device. In this case, referred to as remote debugging, gdb requires two applications – a server running on the target and an attached client on the x86 host.

This is initiated by first executing the server program, `gdbserver`, on the target board. The first argument to `gdbserver` is the IP address and port that the `gdbserver` will listen for a connection on. Note that this is the IP address of the target board. It may seem unnecessary for the server program to specify the IP address of its own system, but recall that some systems have multiple IP connections, such as a hardwired Ethernet and a wireless 802.11 connection, so it is necessary to specify which IP address to listen on.

Once the server is running on the target board, the GNU debugger is launched on the host computer using the `gdb` command. Within the `gdb` application, the user calls the “target remote” command to attach to a remote client. The parameter to the “target remote” command is the same IP address and port as was specified when launching the `gdbserver` on the target. After executing this command `gdb` on the host is attached to `gdbserver` on the target and debugging proceeds as normal.

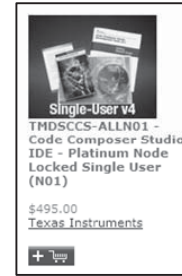
## CCSv5 Licensing & Pricing

### ◆ Licensing

- Wide variety of options (node locked, floating, time based...)
- All versions (full, DSK, free tools) use same image
- Updates readily available via the internet

### ◆ Pricing

- Reasonable pricing – includes FREE options noted below
- Annual subscription – \$99 (\$149 for floating)



| Item                      | Description                             | Price     | Annual |
|---------------------------|---|-----------|--------|
| Platinum Eval Tools       | Full tools with 120 day limit (all EMU) | FREE      |        |
| Platinum Bundle           | EVM, sim, XDS100 use                    | FREE ☺    |        |
| Platinum Node Lock        | Full tools tied to a machine            | \$495 (1) | \$99   |
| Platinum Floating         | Full tools shared across machines       | \$795 (1) | \$159  |
| Microcontroller Core      | MSP/C2000 code size limited             | FREE      |        |
| Microcontroller Node Lock | MSP/C2000                               | \$445     | \$99   |



☺ - recommended option: purchase Dev Kit, use XDS100v1-2, & Free CCSv5

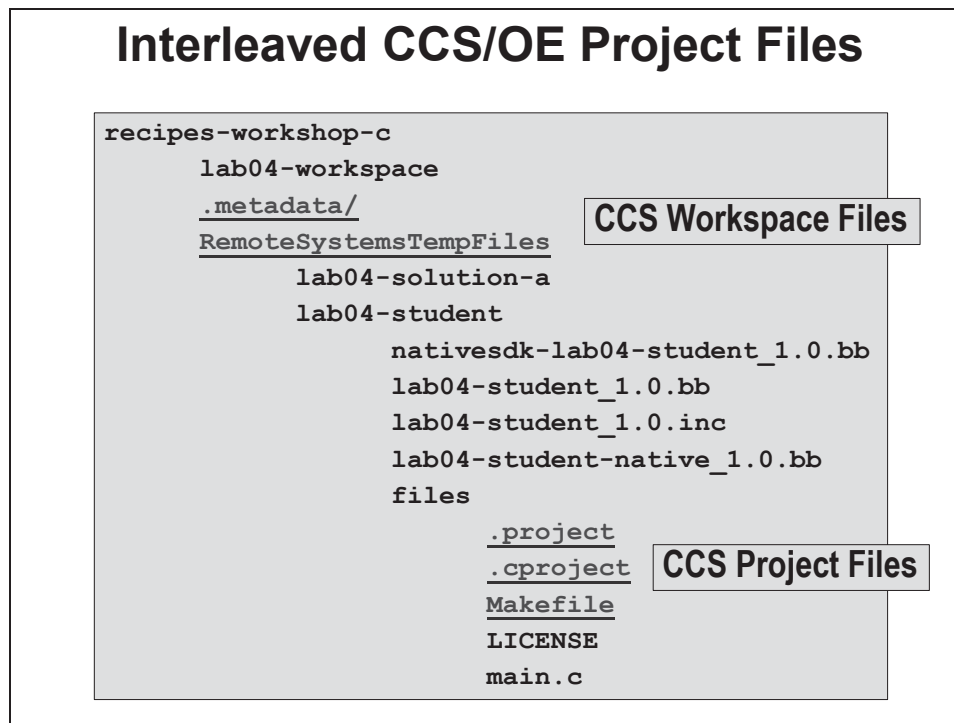
20

Note that the free license of Code Composer Studio allows building programs of any code size and debugging using gdb. If the user requires JTAG-based debugging, a platinum license is required. Node-locked licenses can only be used on a single machine and cost \$495 to purchase, which includes a 1-year upgrade subscription. If the developer wishes to continue receiving updates after the first year, a subscription may be purchased for 20% of the base price or \$99.

There is also an option for a floating license. Floating licenses may be shared across multiple machines. If a company has 20 developers and purchases 10 floating licenses, any 10 developers may open CCS on their machine, but when an 11<sup>th</sup> developer attempts to open CCS they will receive a message that all licenses are in use.

Note that it is possible to mix free and platinum licenses. A team of developers may wish to all install the free CCS and have a small number of floating platinum licenses for the occasions in which JTAG debugging is required.

## CCS-Compatible Project for OpenEmbedded



Unfortunately, eclipse, and therefore CCS, does not currently have native support for bitbake builds. The good news is that eclipse does support user-written makefiles, which are flexible enough to set up bitbake build projects.

A little thought is required to merge the two systems correctly. This is because the bitbake tool expects local files to be located within a subfolder of the bitbake recipe directory, and Code Composer Studio expects files to be located within the project directory, which is generally placed within the workspace directory.

A good layout for a ccs/bitbake project, then, is to place CCS workspaces at the top level of the bitbake repository. The individual bitbake package folders are then placed within these workspaces. These package folders store the application source files in a folder named “files,” which also contains the CCS project files and the makefile used to build the application using bitbake.

## Interleaved CCS/OE Project Files

```

recipes-workshop-c
  lab04-workspace
    .metadata/
      RemoteSystemsTempFiles
        lab04-solution-a
        lab04-student
          nativesdk-lab04-student 1.0.bb
          lab04-student 1.0.bb
          lab04-student 1.0.inc
          lab04-student-native 1.0.bb
          files
            .project
            .cproject
            Makefile
            LICENSE
            main.c

```

**Bitbake Recipe Files**

At the top level of the CCS workspace are all of the bitbake package directories for the packages associated with that workspace. For this workshop, the student project as well as solutions for a given lab exercise are located in the same workspace, and there is a different workspace for each lab.

Within one of these package directories, such as “lab04-student” in this slide, are the individual recipe files. This workshop contains recipes to build each lab exercise under three class types: the base class type, which builds for the Sitara target board, the native class type, which builds for the x86 host, and the nativesdk type, which not only builds for the x86 host, but packages the result into the software development kit installation binary. Bitbake recipes (ending in .bb extension) are provided for each of these three builds, and the fourth file is an include file, which is shared between all three of these recipes and contains the components of the recipe which are common across all three builds.

The slide also highlights the “LICENSE” file, which is contained in the source files directory. The LICENSE file is required by bitbake. This file is empty as this workshop is open-source, but even in this case an empty LICENSE file is required by the bitbake tool.

## Interleaved CCS/OE Project Files

```
recipes-workshop-c
  lab04-workspace
  .metadata/
  RemoteSystemsTempFiles
  lab04-solution-a
  lab04-student
    nativesdk-lab04-student_1.0.bb
    lab04-student_1.0.bb
    lab04-student_1.0.inc
    lab04-student-native_1.0.bb
  files
    .project
    .cproject
    Makefile
    LICENSE
    main.c
```

**CCS Makefile**

Included with the bitbake recipe source files is a makefile script. Bitbake does not require this script in order to build the application, but CCS does.

Since CCS does not currently support a bitbake build profile, it needs a makefile to invoke the bitbake utility to build the recipe. As will be shown on the next slide, this makefile does not directly invoke a compiler, assembler or linker; instead, when the target rule is invoked, it calls bitbake to build the recipe. This guarantees that the same compiler, assembler and linker tools, as well as compatible flag options, are used to build the application as are used to build the Linux distribution that the application will run within. As discussed in the previous module, this provides greater compatibility as well as simpler porting of the application between Linux distributions and even to new hardware platforms.

## CCS Compatible Makefile

```
include ../../../../common.make
PROJECT := $(notdir $(shell cd ../; pwd))

.phony all
all: target

FILES := $(wildcard *.c) $(wildcard *.h)
.phony target
target: $(FILES)
    $(CLEAN) $(PROJECT)
    $(BITBAKE) $(PROJECT)
    cp $(BB_BINDIR)/$(PROJECT) ./$(PROJECT)
    cp $(BB_BINDIR)/.debug/$(PROJECT) ./debug/$(PROJECT)

.phony clean
clean:
    $(CLEAN) $(PROJECT)
```

This is the makefile that is used to build the application packages used as lab exercises in this workshop. The first line includes a common set of definitions from the file “common.make,” which specifies the commands to clean and build the project as well as defining the directory where bitbake places the binaries it builds.

The next line defines the PROJECT variable using the name of the directory that is one level up from the “files” directory that the makefile is stored in. This requires that any bitbake package that uses this makefile adheres to this naming structure.

The target rule invokes bitbake in order to build the application for the target. Note that the first step cleans the sstate cache from the bitbake package. The sstate cache keeps track of which build tasks have been successfully completed for the package. Unlike makefile, bitbake recipes do not check the timestamp of source files. Instead, once a package is built, it will not be rebuilt unless either the package revision (PR) variable is updated or the sstate cache is cleaned.

The target build rule uses the wildcard function to specify all C source (.c) and header (.h) files as dependencies to the target so that the makefile will only force a rebuild of the application package if one of these source files changes.

The final two commands in the target rule copy the application binary and debug symbols from the bitbake deploy directory into the CCS project directory, which is just the directory in which the makefile is located.

## common.make

```
TARGET := am335x-evm
OE_BUILD_DIR := $PWD/../../../../../../build

CLEAN = cd $(OE_BUILD_DIR); source conf/setenv;
      MACHINE=$(TARGET) bitbake -c cleansstate
BITBAKE = cd $(OE_BUILD_DIR); source conf/setenv;
        MACHINE=$(TARGET) bitbake
WORKDIR = $(OE_BUILD_DIR)/arago-tmp-external-linaro-toolchain/
        work/cortexa8hf-vfp-neon-3.8-oe-linux-gnueabi/
BB_BINDIR = $(WORKDIR)/package/home/root/workshop
```

**common.make provides system paths so that the makefile can find what it needs within the bitbake build structure.**

The “common.make” file defines the locations of the various directories associated with the bitbake tool. These directories are referenced off of the “OE\_BUILD\_DIR,” which is assumed to be located six levels up from the CCS project within the subdirectory “build.” This is true if the workshop overlay layer is installed ontop of the Arago bitbake files as specified in the setup document.

The “common.make” file defines macros to clean and build the bitbake package as well as creating variables to reference the deployment directory where the application binaries will be placed by bitbake once the application has been built.



## Interleaved CCS/OE Project Files

```
recipes-workshop-c
  lab04-workspace
  .metadata/
  RemoteSystemsTempFiles
    lab04-solution-a
    lab04-student
      nativesdk-lab04-student 1.0.bb
      lab04-student 1.0.bb
      lab04-student 1.0.inc
      lab04-student-native 1.0.bb
      files
        .project
        .cproject
        Makefile
        LICENSE
        main.c
```

**Bitbake Recipe Files**

The workshop uses three bitbake recipes for each lab student exercise and solution. These are “nativesdk-(package name)\_1.0.bb,” which builds the application for the x86 host to be placed within the software development kit, “(package name)\_1.0.bb,” which builds the application for the Sitara target board, and “(package name)-native\_1.0.bb,” which builds the application for the x86 host without packaging into the sdk.

Note that the names of these recipe files determine the name and version of the associated package that they build according to the format “(package name)\_(package version).bb” While bitbake does not require that the directory in which these files are located match the package name, this is common practice. Furthermore, the CCS makefile assumes that this is the case and will not work properly unless this directory name and the package name match.

## Workshop Recipe Files

### lab04-student 1.0.inc

Bitbake include file. This file is included by each of the three Bitbake recipes below and contains definitions that are common across the three recipe types.

### lab04-student 1.0.bb

This standard Bitbake recipe conforms to the base class. (“inherit” statement is not required for the base class.) The base class will build the application to run on the `$(MACHINE)` platform. This recipe is utilized by the CCS Makefile to build applications to be run and debugged on the AM335x

### lab04-student-native 1.0.bb

This Bitbake recipe inherits the “native” class. The native class will build the application to run on the `$(BUILD_SYS)` platform

### nativesdk-lab04-student 1.0.bb

This Bitbake recipe inherits the “nativesdk” class. The nativesdk class will build the application to run on the `$(BUILD_SYS)` platform and will additionally package the application into the Arago SDK.

## One Last Detail

```

recipes-workshop-c
  lab04-workspace
  .metadata/
  RemoteSystemsTempFiles
    lab04-solution-a
    lab04-student
      nativesdk-lab04-student_1.0.bb
      lab04-student_1.0.bb
      lab04-student_1.0.inc
      lab04-student-native_1.0.bb
      files
        .project
        .cproject
        Makefile
        LICENSE
        main.c

```

**Source Files(s)**

A final detail should be mentioned. The source file “main.c” has been listed as shown above on the previous slides, but is actually arranged within a number of subdirectories in the bitbake packages of the workshop overlay. The reason for this is that when bitbake embeds debugging information into the applications that it builds, it references the source files used to build the application according to the absolute path of their location on the target filesystem.

## One Last Detail

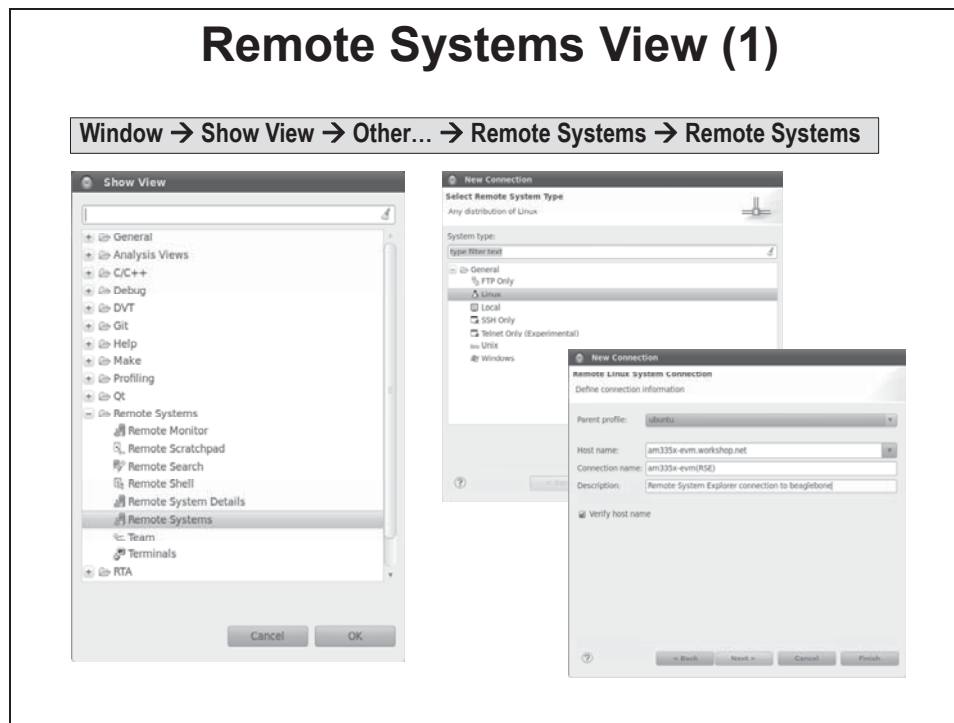
```
files
    .project
    .cproject
    Makefile
    LICENSE
    usr/src/debug/lab04-student/
        1.0-r0.0/main.c
```

**Within the files folder of the CCS project, all source files need to be placed within subdirectories that reflect their location within the filesystem of the target board.**

**This is because the debugging information placed within the binary executable references the files by their location on the target board. In order for CCS to find the source files during debugging they need to contain this directory structure.**

By default, bitbake places application source files on the target filesystem at the location “usr/src/debug/(project name)/(project version)” and this is the path that is specified in the debugging symbol file. By placing the file within a matching subdirectory path within the CCS project, CCS is able to locate the file during debugging sessions.

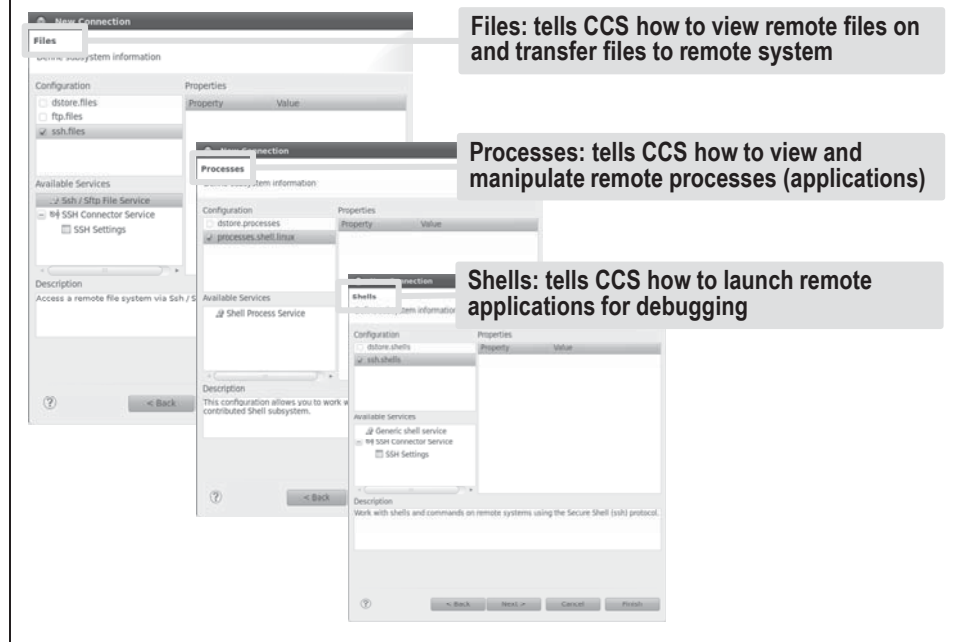
## Setting up a CCSv6 GDB Session



The first step in setting up the GNU debugger in eclipse is to create a new connection to the target board in the remote systems view. The remote systems view window is brought up from the “Show View” selection under the “Window” pulldown menu. If the remote systems view window has been brought up recently it will be cached on this pulldown menu, otherwise select “other...” for the full listing of windows and select the “Remote Systems” window from the “Remote Systems” folder.

From the remote systems window, launch the new connection wizard by pressing add the connection icon (three-way pipe with an addition sign overlaying it.) The first two windows of the new connection window are shown on the tight-hand side of this slide. In the first window the connection type is selected. The recommended type for this configuration is “Linux.” On the next window, the host name or IP address of the target is specified, and the connection is given a name. The description field is optional.

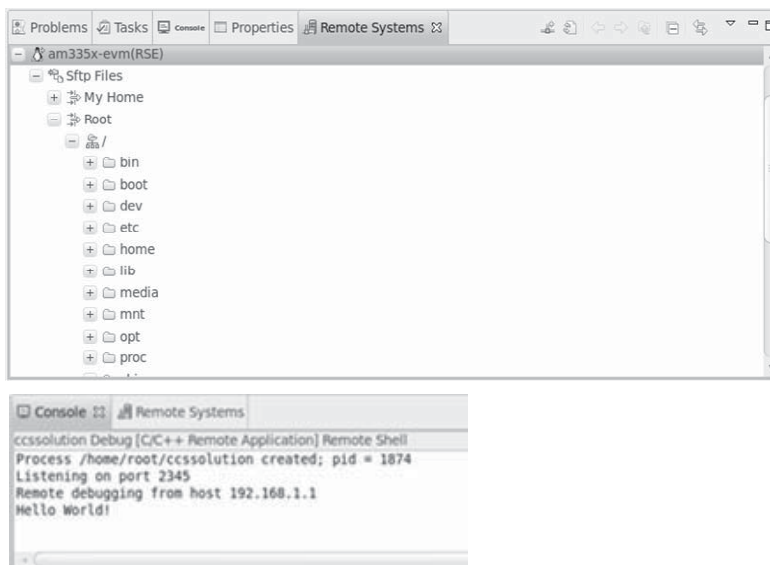
## Remote Systems View (2)



In the final three windows of the wizard, select the protocols that eclipse will use to communicate with the target. Any of the connection types will work so long as the target is running the corresponding server.

TI's recommendations for the connection are "ssh.files" for file transfer, "processes.shell.linux" to view or halt running processes on the target board, and "ssh.shells" as the terminal shell program.

## Remote System View – Files / Terminal



After successful configuration of the remote system view connection, a new connection will appear in the remote system view tab with the name specified on the second window of the new connection wizard. Expanding the connection, the “Sftp Files” branch, and then the “Root” branch will provide a view of the filesystem of the remote target. Double-clicking a file in the tree will open the file inside of the CCS editor, and if the file is modified and saved, the modification is updated on the target.

## Setting Up CCSv5 Application Debug

**Run → Debug Configurations → New Launch Configuration**

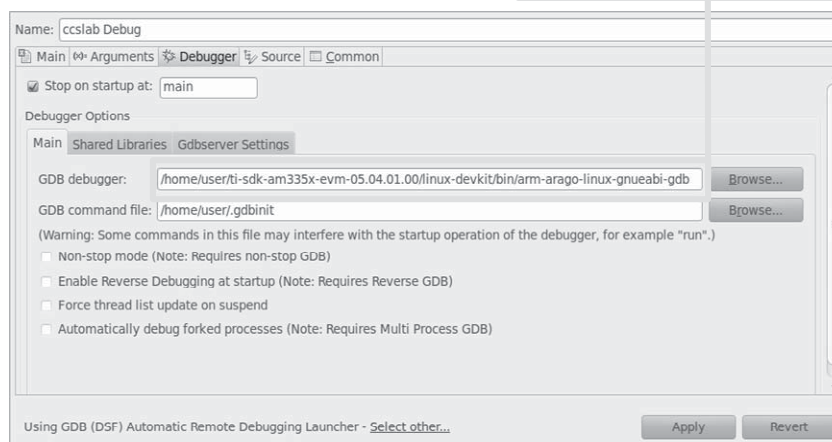
**Use remote system explorer to launch remote gdbserver**

The next step is to set up the actual debugging connection. Select “Debug Configurations” from the “Run” pulldown menu. Many of the fields will be filled in with default values, but you will need to selection the connection to the remote target from the pulldown menu next to the “Connection” label.

You will also need to specify the “Remote Absolute File Path for C/C++ Application.” The path specified needs to be a valid path on the target board. The path specified will be the location where CCS places the application binary when it is copied down to the target board before executing within gdbserver.

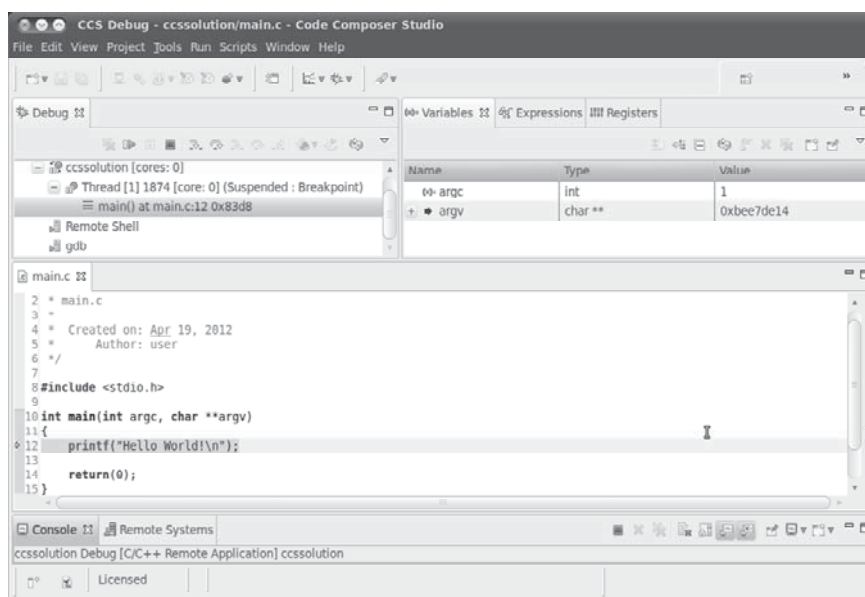
## Setting Up CCSv5 Application Debug

Specify version of Gnu Debugger to use



On the debugger tab you will need to specify the cross-gdb debugger on your system. The default value is “gdb,” which is the default gdb debugger on the host computer. Generally this is an x86 debugger. You will need to replace this with the location of the cross gdb debugger that is installed within the software development kit.

## CCSv5 Attached Remote Debugger



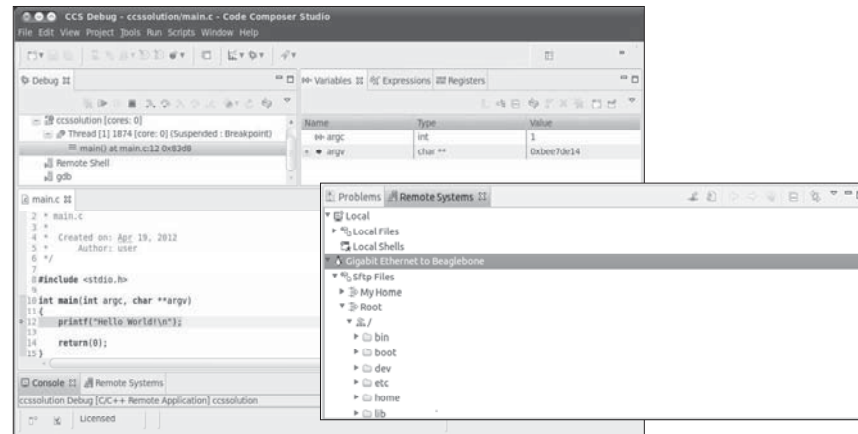
A successful launch of gdb/gdbserver will appear as on the screen capture shown. By default CCS will halt on the first line of main.



## Lab 4: Application Build and Debug

### Lab 4: Application Build and Debug

- A. Test Secure Shell (ssh) protocol
- B. Create Project
- C. Set up CCS Remote System Explorer Connection
- D. Set up CCS Debug Configuration



(Page intentionally left blank)

# Module 05: Linux Files

---

## Introduction

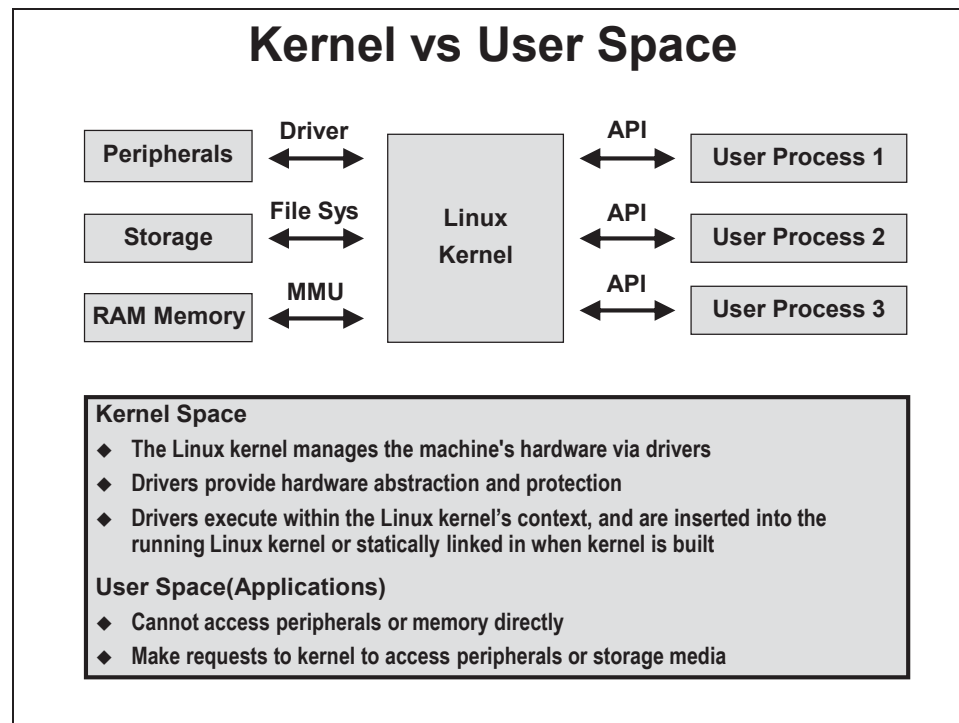
The Linux operating system has a wide range of filesystem support for many different block storage devices. The ability to store and manipulate files on NAND and NOR flash, multimedia and secure digital cards, hard disk drives and other media is important for many systems.

This section begins with an overview of Linux drivers that is applicable both to storage media drivers and device drivers. This module then continues to two case studies: the Arago NAND flash driver and GPIO driver. Though GPIO is a peripheral, the Arago GPIO driver is manipulated via virtual files and programming GPIO is therefore more akin to file manipulation than standard device driver manipulation. The next chapter will then focus on device drivers.

## Module Topics

|  |             |
|--|-------------|
| <b>Module 05: Linux Files .....</b>        | <b>5-1</b>  |
| <i>Module Topics.....</i>                  | <i>5-2</i>  |
| <i>Driver Basics .....</i>                 | <i>5-3</i>  |
| <i>NAND Flash Driver .....</i>             | <i>5-15</i> |
| <i>GPIO.....</i>                           | <i>5-18</i> |
| <i>Lab 5: GPIO via Virtual Files .....</i> | <i>5-22</i> |

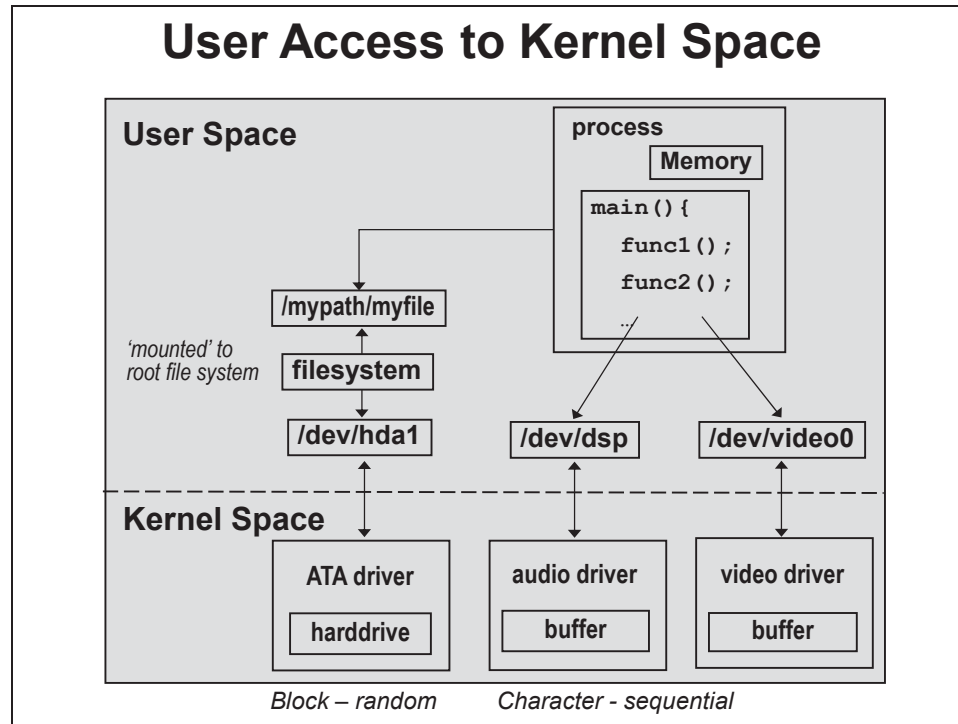
# Driver Basics



A key concept in the design of the Linux operating system is the separation of user applications, also known as processes, from the underlying hardware. User-space applications are disallowed from accessing peripheral registers, storage media or even RAM memory directly. Instead, the hardware is accessed via kernel drivers, and RAM memory is gated by the memory management unit, with applications operating on virtual addresses.

This separation provides robustness. If it is assumed that the Linux kernel is operating correctly (i.e. the kernel is a trusted entity in the system) then allowing only the kernel to interact with underlying hardware keeps applications from accidentally or maliciously mis-configuring hardware peripherals and placing them in unknown or non-functioning states.

Removing hardware-specific references from application code also provides portability. By localizing hardware-specific code to kernel drivers, only these drivers need to be modified in order to port a system from one hardware platform to another. Applications access a set of driver API that is consistent across hardware platforms, allowing well-written applications to be moved from one platform to another with little or no modification to the source code.



Applications do, however, require the ability to access data that is captured by hardware peripherals, as well as the ability to drive peripherals with output. Since the peripheral registers are accessible only by the Linux Kernel (i.e. they exist within “Kernel Space” as shown on the diagram), only the kernel is able to collect data streams as they are captured by these peripherals. Linux therefore requires a mechanism for these data streams to be transferred from kernel space to the memory space within which the various user applications reside, shown as “User Space” on the slide.

This transfer of data streams is handled via device nodes, which are also known as virtual files. Device nodes exist within the root filesystem, though they are not true files. When a user reads from a device node, the kernel copies the data stream captured by the underlying driver into the application memory space. When a user writes to a device node, the kernel correspondingly copies the data stream provided by the application into the data buffers of the driver, which are eventually output via the underlying hardware.

This slide shows two types of device drivers – character drivers and block drivers. Character drivers access data as a stream, in a first-in-first-out (FIFO) basis. This means that when the device node of the driver is opened, “time zero” is established and the device begins capturing. Data may then be read out of the device node in the order that it was captured from time zero, i.e. the first read is the value sampled at “time one” and then “time two” etc. Note that the original streaming drivers were character drivers used for terminals, but that this driver basis has been extended over time and now is used for many drivers, including audio and video drivers, which are Linux “character” drivers even though this may seem a misnomer.

The second type of drivers are block drivers. Block drivers allow out-of-order access (i.e. “Random Access” in computer terms) of data. These drivers are typically used for mass-storage devices where it does not make sense to read data only in sequential order. For instance when a user accesses a 1 terabyte hard disk drive, that user does not typically wish to begin at byte one of the device and read through 1 trillion bytes. Instead there is usually a specific location of the

harddrive that the user wishes to access, and generally the data on the device is managed by a filesystem which organizes data by files, directories, etc. This is why user applications will generally access block devices via a filesystem instead of accessing the device node directly. A good litmus test to determine if a driver is likely a character device or a block device is that devices that make sense to mount a filesystem onto, such as harddrives, MMC/SD cards, NAND and NOR flash and RAM memory are typically accessed via block drivers, and most other devices are accessed via character drivers.

## Four Steps to Accessing Drivers

- 1. Load the driver's code into the kernel (insmod or static)**
- 2. Create a virtual file to reference the driver using mknod**
- 3. Mount block drivers using a filesystem (block drivers only)**
- 4. Access resources using open, read, write and close**

This slide shows the recipe for accessing peripherals via an application. Firstly, the driver must be loaded into the kernel, either as a static driver that is built into the kernel binary, or as a dynamically loadable module that is loaded into the kernel at runtime using either the “insmod” or “modprobe” commands.

Next, a virtual file must be created in the filesystem to access the driver, and, if the device is a block driver, the user will also typically mount the device node into the root filesystem as per step 3.

Finally, the user can access the data associated with the driver using open, read, write and close (or in the case of files on the filesystem of a block device, via fopen, fread, fwrite and fclose.)

Note that steps 1 and 2 have often been completed for the user for most drivers in the majority of Linux distributions. For instance, the Arago distribution for a given hardware platform such as the Beaglebone will have drivers statically built into the kernel for all of the peripherals available on the board, and device nodes for these drivers are already set up in the root filesystem.

We will take a closer look now at how each of these four steps is accomplished via standard Linux utilities, beginning with step 1.

## Kernel Object Modules

How to add modules to Linux Kernel:

**1. Static (built-in)**

**Linux Kernel**

audio

fbdev

httpd

v4l2

nfsd

dsp

ext3

Kernel Module Examples:

|              |                         |
|--------------|-------------------------|
| <b>fbdev</b> | frame buffer dev        |
| <b>v4l2</b>  | video for linux 2       |
| <b>nfsd</b>  | network file server dev |
| <b>dsp</b>   | oss digital sound proc. |
| <b>audio</b> | alsa audio driver       |

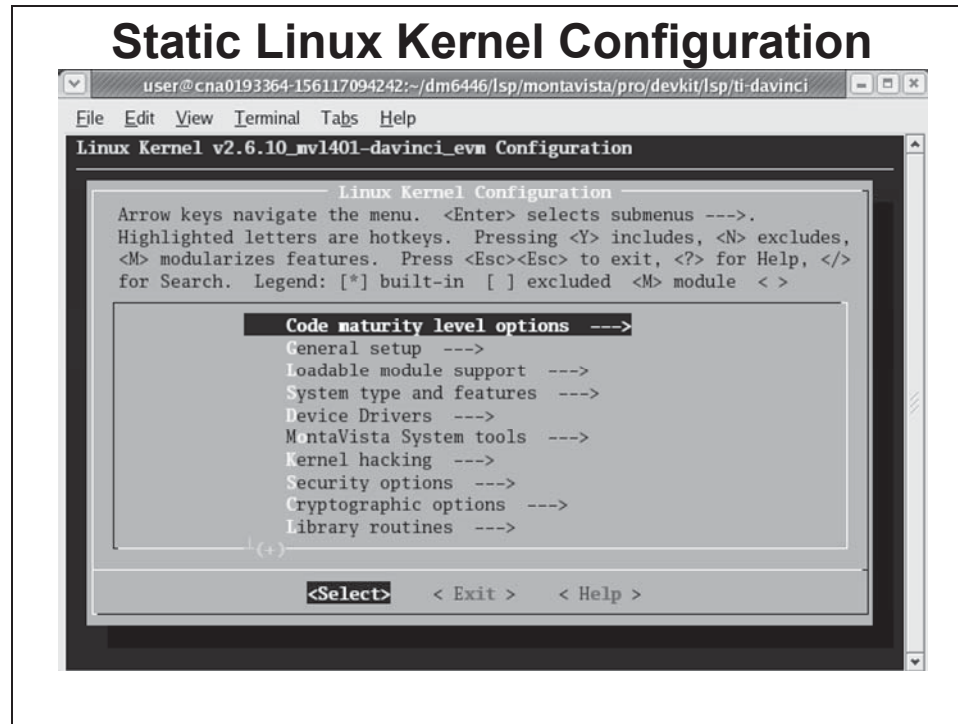
- ♦ Linux Kernel source code is broken into individual modules
- ♦ Only those parts of the kernel that are needed are built in

Change static configuration using...

The Linux kernel is modularized to allow configuration of modules the user requires without having to add unnecessary code for features that the system does not use. Some of the most common types of modules are drivers modules and filesystem modules.

The simplest means of adding a driver or filesystem module into the kernel is to add the module statically when building the kernel binary.





The Linux kernel makefile contains a utility called “menuconfig” that may be accessed via the command “make menuconfig” from the terminal.

This slide shows the top-level screen of the menuconfig utility. Modules are organized into categories. The user can use the arrows to select, for instance, the “Device Drivers” category, and then a subcategory such as “multimedia devices.” After descending the applicable categories, the menuconfig screen shows the individual modules grouped into that subcategory, and the user may select the module and press either “y” to statically build the module into the kernel, “n” to remove the module altogether, or “m” to create a dynamically loadable module.

## Kernel Object Modules

How to add modules to Linux Kernel:

**1. Static (built-in)**

**Linux Kernel**

oss

fbdev

httpd

v4l2

nfsd

dsp

ext3

Kernel Module Examples:

**fbdev** frame buffer dev  
**v4l2** video for linux 2  
**nfsd** network file server dev  
**dsp** oss digital sound proc.  
**audio** alsa audio driver

- ♦ Linux Kernel source code is broken into individual modules
- ♦ Only those parts of the kernel that are needed are built in

---

**2. Dynamic (modprobe)**

*.ko = kernel object*

```
# modprobe <mod_name> .ko [mod_properties]
```

- ♦ Use modprobe command to dynamically add modules into the Linux kernel
- ♦ Keep statically built kernel small (to reduce size or boot-up time), then add functionality later with modprobe
- ♦ modprobe is also handy when developing kernel modules

Dynamically loadable modules are placed in their own file and given the “.ko” file extension, which is an abbreviation for “kernel object.” These kernel object files are stored in the root filesystem and may be loaded into the kernel at runtime using either the “insmod” or “modprobe” commands. The “modprobe” command is used more frequently due to the fact that it searches module dependencies and will load dependent modules recursively as necessary to support any module that the user adds into the kernel.

## Examining The Steps in More Detail...

1. Load the driver’s code into the kernel (insmod or static)
2. Create a virtual file to reference the driver using mknod
3. Mount block drivers using a filesystem (block drivers only)
4. Access resources using open, read, write and close

## Linux Driver Registration

```
# mknod <name> <type> <major> <minor>

<name>:  Node name (i.e. virtual file name)
<type>:  b      block
         c      character
<major>:  Major number for the driver
<minor>:  Minor number for the driver
```

```
Example:  mknod  /dev/fb/3  c  29  3
Usage:    Fd = open("/dev/fb/3", O_RDWR);
```

- ◆ Register new drivers with *mknod* (i.e. Make Node) command.
- ◆ **Major number** determines which driver is used (the name does not affect which driver is used). Most devices have number assigned by Linux community.
- ◆ **Minor number** is significant for some drivers; it could denote instance of given driver, or in our example, it refers to a specific buffer in the FBdev driver.

Once the driver module has been either statically built into the kernel or dynamically loaded, the user needs to create a device node to reference the driver. Linux provides the “mknod” (MaKe NODe) utility for this purpose.

The mknod command takes four parameters. The first parameter is the name of the device node that will be created. The device node should be placed in the “/dev” directory, and, while technically a device node can have any name the user chooses, it is generally best to follow the naming conventions that the Linux community has established since these will be the device node names that are expected by applications developed throughout the community.

The next parameter indicates whether the driver to which the device node interfaces is a block driver or character driver. This is important because the kernel will use a different set of API to interface to a block driver than a character driver, and which API is used is determined by this setting.

The final two parameters to mknod are the major and minor numbers. The major number is a unique integer number that is assigned to any driver that is loaded into the Linux kernel. Most drivers use statically assigned major numbers that are determined by the community, so the simplest method for determining a driver’s major number is via community documentation. Also, assigned major numbers are listed in the “/proc/devices” (text) file and can be determined by viewing this file using “cat” or a similar command.

The minor number may be used in various ways, so the user should consult the documentation of the driver they are using in order to determine the correct minor number setting. Usually this number is used to manage instances of the driver. For instance, the AM33658 has six hardware UART ports. The same driver can be used to control all six UARTS, but each physical UART needs its own device node, so the device nodes for these UARTS will all have the same major number (corresponding to the UART driver used) but will have unique minor numbers.

## Linux Device Registration

- ◆ Linux devices are registered in /dev directory
- ◆ Two ways to view registered devices:
  - cat /proc/devices
  - ls -lsa command (as shown below) to list available drivers

```

/ # cd /dev
/dev # ls -lsa
0 brw-rw---- 1 root disk 0, 0 Jun 24 2004 /dev/hda
0 crw-rw---- 1 root uucp 4, 64 Mar 8 2004 /dev/ttyS0
0 crw----- 1 user root 14, 3 Jun 24 2004 /dev/dsp
0 crw----- 1 user root 29, 0 Jun 24 2004 /dev/fb/0
0 crw----- 1 user root 29, 1 Jun 24 2004 /dev/fb/1
  
```

Annotations:

- Permissions (user,group,all)
- Major number
- Minor number
- Name
- /dev directory
- block vs char

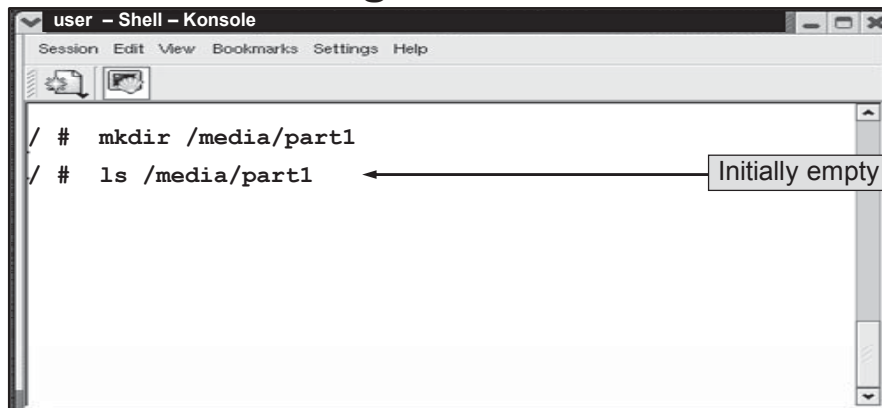
Often a developer is using a Linux distribution that was provided to them, such as the Arago open-source distribution that is maintained by Texas Instruments. In this case, a set of drivers has usually been statically built into the Linux kernel, in some cases all of the drivers that the user will require. And if a distribution provides static drivers in the provided kernel, it will typically provide the corresponding device nodes in the root filesystem.

A first step to using Linux drivers from a pre-provided distribution is often to view the drivers that the distribution has already statically loaded and registered device nodes for. This can be done by viewing the “/proc/devices” system file to see which drivers are loaded and then listing the contents of the “/dev” directory to see which device nodes are provided. The “/proc/devices” system file shows the major number of each driver, and listing the contents of the “/dev” directory using the “-lsa” options provides the major number associated with each device node, so that it is simple to determine the driver associated with each device node.

## Examining The Steps in More Detail...

1. Load the driver's code into the kernel (insmod or static)
2. Create a virtual file to reference the driver using mknod
3. Mount block drivers using a filesystem (block drivers only)
4. Access resources using open, read, write and close

## Mounting Block Devices



```
user - Shell - Konsole
Session Edit View Bookmarks Settings Help

/ # mkdir /media/part1
/ # ls /media/part1
```

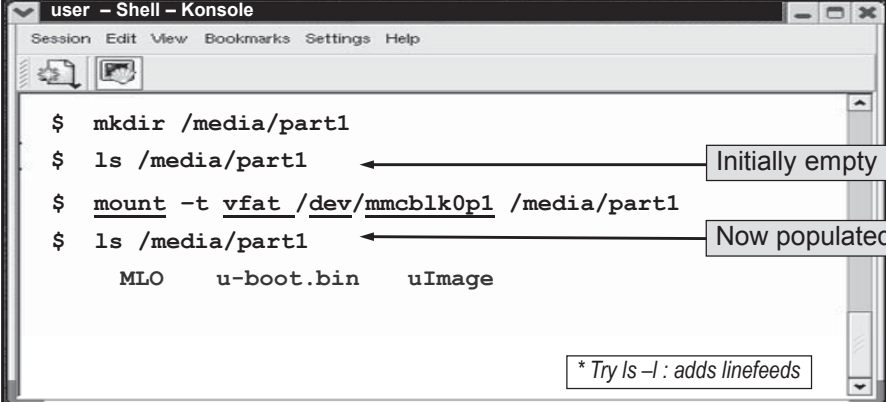
Initially empty

- ◆ Mounting a block driver into the filesystem gives access to the files on the device as a new directory
- ◆ Easy manipulation of flash, hard drive, compact flash and other storage media
- ◆ Use mkfs.ext2, mkfs.jffs2, etc. to format a device with a given filesystem

While it is possible for an application to access the device node of a block device directly, the API set used to interact with block device nodes are generally more low-level than the application author will require. By mounting the device into the root filesystem, the application author will gain access to the contents of the device via a filesystem, which will manage the data into files and directories.

The first step in mounting the device is to create an empty directory.

## Mounting Block Devices



```
$ mkdir /media/part1
$ ls /media/part1
$ mount -t vfat /dev/mmcblk0p1 /media/part1
$ ls /media/part1
    MLO    u-boot.bin  uImage
```

- ◆ Unlike Windows, there is only one filesystem – therefore you must mount to a mount point (i.e. empty directory) in the root filesystem
- ◆ Easy manipulation of flash, hard drive, compact flash and other storage media
- ◆ Use `mkfs.ext2`, `mkfs.jffs2`, etc. to format a device with a given filesystem
- ◆ The above example shows mounting an external harddrive into the root filesystem

The contents of the block device are then mounted into the root filesystem at this location. The mount command is used. It takes an optional “-t” argument to specify the type of filesystem that the device is formatted with. If this option is left off, Linux can usually determine the filesystem type automatically. The next parameter to the mount command is the device node corresponding to the device that is to be mounted. The final parameter is the mount point within the root filesystem to mount the contents of the device. In this example, the mountpoint is “/media/part1,” the empty directory that was previously created.

After mounting the block device, the contents of the device appear within the root filesystem at the mount point. Note that the contents of the block device are not actually copied into the root filesystem, they are simply shadowed at this location. If the user were to unmount the device using the “umount” command, the mount-point directory would be empty again.

## Some Common Filesystem Types

### Harddrive File systems:

|             |   |
|-------------|---|
| <b>ext2</b> | Common general-purpose filesystem   |
| <b>ext3</b> | Journalled filesystem -<br>Similar to ext2, but more robust against unexpected power-down |
| <b>vfat</b> | Windows "File Allocation Table" filesystem  |

### Memory File systems:

|               |   |
|---------------|---|
| <b>jffs2</b>  | Journaling flash filesystem (NOR flash)   |
| <b>yaffs</b>  | yet another flash filesystem (NAND flash) |
| <b>ramfs</b>  | Filesystem for RAM                        |
| <b>cramfs</b> | Compressed RAM filesystem                 |

### Network File systems:

|              |                                    |
|--------------|------------------------------------|
| <b>nfs</b>   | Share a remote linux filesystem    |
| <b>smbfs</b> | Share a remote Windows® filesystem |

Linux supports quite a few filesystems, many of which are optimized for specific media. A list of some of the more commonly used filesystems is provided in the slide above, categorized by the physical media they are generally used to manage.

## Examining The Steps in More Detail...

1. Load the driver's code into the kernel (insmod or static)
2. Create a virtual file to reference the driver using mknod
3. Mount block drivers using a filesystem (block drivers only)
4. Access resources using open, read, write and close

Once character drivers have device nodes to access and block devices are mounted into the root filesystem, their corresponding resources may be accessed using the standard open, read, write and close API.

## Accessing Files

Manipulating files from within user programs is as simple as...

```

myFileFd = fopen("/mnt/harddrive/myfile", "rw");
fread ( aMyBuf, sizeof(int), len, myFileFd );
fwrite( aMyBuf, sizeof(int), len, myFileFd );
fclose( myFileFd );

```

Additionally, use `fprintf` and `fscanf` for more feature-rich file read and write capability

In this module, we will focus on accessing the content of block devices. Since these devices mount into the root filesystem and are organized into files, the “f” version of the open, read, write and close calls are typically used to access their contents.

The first step is to open a file using “fopen.” The function takes the name of the file to open, either as an absolute or relative path, and the second parameter specifies the read and write permissions with which to open the file. The function returns a file handle that is used to reference the opened file in subsequent calls. (Note that more than one file may be opened at a time, so it is necessary to maintain a reference to each opened file.)

The `fread` call reads the contents of the file and stores them into an array inside of the application memory. The first parameter to the call is a pointer to the array. The next two fields specify the block size and number of blocks to read from the file into the array. The blocksize (specified in bytes) will just be multiplied with the number of blocks to determine the total number of bytes to read. The final parameter is the file handle to read from.

The `fwrite` call works much as the `fread` call but in the other direction.

Finally, when the file is no longer needed, it is good practice to close the file using the `fclose` call. This call requires only the file handle.

This is not an exhaustive list of function that may be used to access files. Other functions such as `fseek`, `fprintf` and `fscanf` provide additional functionality, but the basic `fopen`, `fread`, `fwrite` and `fclose` operations will be sufficient for this basic introduction.



# NAND Flash Driver

## NAND Partitions

The Memory Technology Device (mtd) driver defines seven fixed partitions of NAND. Unlike most block devices, these partitions cannot be changed without modifying and rebuilding the driver source.

```
# cat /proc/mtd

dev:   size      erasesize  name
mtd0:  00020000  00020000  "SPL"
mtd1:  00020000  00020000  "SPL.backup1"
mtd2:  00020000  00020000  "SPL.backup2"
mtd3:  00020000  00020000  "SPL.backup3"
mtd4:  001e0000  00020000  "U-Boot"
mtd5:  00020000  00020000  "U-Boot Env"
mtd6:  00500000  00020000  "Kernel"
mtd7:  0f880000  00020000  "File System"
```

NAND flash is accessed with the MTD (memory technology device) driver. This driver supports partitions, although these partitions are not implemented via a partition table resident on the device the way that partitions on a harddrive or multimedia card would be. Instead, the partitions are maintained by the driver and are modified either by recompiling the driver or by supplying the “mtdparts” command-line argument to the Linux kernel at boottime.

The current settings of the partitions, including size and name, can be viewed by printing out the system file “/proc/mtd”

This slide shows the default partition settings for the AM335x MTD driver on the Arago distribution. The driver has been configured to support a bootable image and contains four partitions for the Secondary Bootloader (SBL), one partition for the u-boot bootloader, one partition for the u-boot environment variables, one partition for the Linux kernel and one partition for the root filesystem.

The reason that there are four SBL partitions is so that four identical copies of the SBL may be written into the NAND flash. A property of NAND flash is that it is expected to experience physical wearing over time and suffer bit errors that cannot be repaired. Generally these bit errors are compensated for using Forward Error Detection and Forward Error Correction. The ROM bootloader on the AM335x, however, does not have the ability to perform Forward Error Correction and as such requires multiple copies of the SPL as backups in case one or more copy suffers bit errors.

## Examples of Using NAND

### Erase NAND partition 4

```
# flash_eraseall /dev/mtd4
```

### Write u-boot into NAND partition 4

```
# nandwrite -p /dev/mtd4 u-boot.img
```

### Write a UBI image into NAND partition 7

```
# ubiformat /dev/mtd7 -f ubi.img -O 2048
```

UBI images are created with `mkfs.ubifs` and `ubinize` commands.  
For more information:  
[http://processors.wiki.ti.com/index.php/UBIFS\\_Support](http://processors.wiki.ti.com/index.php/UBIFS_Support)

Before writing to a nand partition, the “flash\_eraseall” command is used to erase all of the erase blocks in the partition. The “flash\_eraseall” command takes the device node corresponding to the partition as its parameter. The device node corresponding to a given partition can be determined by viewing the “/proc/mtd” system file as shown on the previous slide. Recall that FLASH always needs to be erased before it can be written and that a full erase block must be erased.

After the partition has been erased, a file may be written into the partition using the “nandwrite” command. All but the final partition can be written using this command since they require writing only a single file (except the u-boot environment partition which is maintained by uboot and therefore isn’t written by the user at all.)

The seventh and final partition, however, contains the root filesystem, which has many files, and therefore cannot be written using “nandwrite.” The workshop covers two methods for writing the root filesystem into this partition. Firstly, if a ubifs image is available, the entire image may be written into the partition using the `ubiformat` command.

The SDK for the Arago distribution ships with root filesystem images in both a ubifs format and a tape archive (.tar.gz) format. If an image of the desired root filesystem is not available, the “mkfs.ubifs” and “ubinize” commands can be used to create a ubifs image as outlined on the provided wiki page link.

## Mounting a UBI Volume

```
# ubinfo
# mtdinfo

# mkdir /mnt/ubifs
# ubiattach /dev/ubi_ctrl -m 7 -O 2048
# mount -t ubifs ubi0:rootfs /mnt/ubifs
```

ubiattach attaches mtd partition 7 to UBI

-O 2048 indicates NAND flash block size

UBI volumes are specified ubiX:Y

X device number

Y Volume number (ubi0:0) or name (ubi0:rootfs)

The second method for writing the root filesystem is to mount the rootfs partition and write the files directly.

Mounting the NAND partition is the same conceptually as mounting any block device node; however, the procedure is slightly different than the generic procedure outlined in the previous section.

Just as in the generic procedure, begin by creating an empty directory as the mount point. The second step is a new command. Before the mount command can be used, call “ubiattach” as shown. The first parameter “/dev/ubi\_ctrl” is the ubi control node and will not change. The second parameter (-m) indicates the partition number and the final parameter (-O) indicates the block size. These values can be determined using the “ubinfo” and “mtdinfo” commands, which will display various information about the ubi filesystem and mtd driver to the terminal.

The final command is as before, the mount command. The filesystem type should be specified as “ubifs.” Instead of taking a device node as the next parameter, specify the partition using the “ubiX:Y” format where “X” is the device number and “Y” is the partition number. The final parameter indicates the mount point, which, in this case, is the empty directory previously created at “/mnt/ubifs.” Alternately, the partition name can be used instead of the partition number. As before, the needed information is available from the “ubinfo” and “mtdinfo” commands.

## GPIO

### GPIO Driver (Terminal)

#### 1. Export GPIO pin for sysfs usage (Pin 53 in example)

```
# echo 53 > /sys/class/gpio/export
```

#### 2. Change the GPIO pin direction to in/out

```
# echo "out" > /sys/class/gpio/gpio53/direction  
# echo "in" > /sys/class/gpio/gpio53/direction
```

#### 3. Change the value (output) or read the value (input)

```
# echo 1 > /sys/class/gpio/gpio53/value  
# echo 0 > /sys/class/gpio/gpio53/value  
# cat /sys/class/gpio/gpio53/value
```

#### 4. Unexport GPIO pin (when no longer needed)

```
# echo 53 > /sys/class/gpio/unexport
```

The General-Purpose Input/Output pin driver in Linux is controlled via virtual files in the “system filesystem” (sysfs.) The user reads and writes to these virtual files the same as one would write to a standard file, and the result controls the driver.

“sysfs” is a filesystem that was developed to provide an additional interface into drivers via virtual files. Most drivers use sysfs only for configuration and debugging and maintain a device node for data transfer. The GPIO driver is a special case where the entire driver interaction is controlled via sysfs. This provides simpler control, particularly from the terminal command line, but is less efficient.

The first step in accessing a GPIO pin is to export the pin. Since the number of GPIO pins on a device is on the order of one hundred, it would not make sense to maintain a directory with value and direction files for every one by default. Instead, the driver requires you to export the GPIO before it creates these files.

Once the GPIO pin has been exported, the direction is set by writing the string value of “in” or “out” into the direction virtual file in the GPIO directory. If the GPIO is an output, it may be driven high or low by writing a “1” or “0,” respectively, into the “value” virtual file. Note that this is a character – i.e. must be the ASCII value of the “1” or “0” character, not an integer value of 1 or 0! If the GPIO is an input, the value may be sampled by reading from the value virtual file.

When the user is finished accessing the GPIO they may unexport, which will remove the gpio pin directory and the virtual files within it.

## GPIO Driver (C code)

sysfs virtual files can be manipulated from C code in same manner as regular files

```
// echo "out" > /sys/class/gpio/gpio30/direction
pFile = fopen("/sys/class/gpio/gpio30/direction", "w");
fwrite("out", 1, sizeof("out"), pFile);
fclose(pFile);

// echo 1 > /sys/class/gpio/gpio30/value
pFile = fopen("/sys/class/gpio/gpio30/value", "w");
fwrite("1", 1, sizeof("1")-1, pFile);

// cat /sys/class/gpio/gpio30/value
fread(myArray, 1, 1, pFile);
fclose(pFile);
```

**Note: sizeof("1") returns a value 2, includes the string terminating character. This needs to not be written.**

It is fast and simple to manipulate GPIO from the terminal command line with the echo command and redirection as shown on the previous slide. It is only slightly more involved to accomplish the same tasks from a C or C++ application. The virtual files shown are manipulated via the file api discussed in this module. Simply "fopen" the virtual file and then read or write as appropriate.

Note that the "value" virtual file expects to receive a character and not a string. If a string is written instead of a character, it will contain a null termination character. This termination character will be interpreted as the last setting of the GPIO, which will always cause it to drive low!

## leds-gpio driver

**In addition to standard (device-specific) GPIO driver, Arago provides the (platform-specific) leds-gpio driver:**

```
/sys/devices/platform/leds-gpio/leds/am335x:EVM_SK:heartbeat  
/sys/devices/platform/leds-gpio/leds/am335x:EVM_SK:mmc0  
/sys/devices/platform/leds-gpio/leds/am335x:EVM_SK:usr0  
/sys/devices/platform/leds-gpio/leds/am335x:EVM_SK:usr1
```

### 1. Light the usr0 LED

```
# echo 1 > /sys/devices/platform/leds-gpio/leds/am335x\EVM_SK\usr0/brightness
```

Note that “\.” is used to designate “.” in the Linux terminal because “.” is a special character. “\” is called the escape character because it can be prepended to special characters to designate they are simple text.

For GPIO that are attached to LEDs, the leds-gpio driver provides a simpler interface and extended functionality. The driver does not require an export step, so the user may simply write a brightness value to the brightness virtual file in the led directory. The generic Linux driver accepts a value from 0-255 for the brightness of the LED, allowing the user to adjust the LED brightness. The TI leds-gpio driver for the AM335x currently ignores the brightness and turns the LED to full on for any value from 1-255 and full off for 0, so that one would generally write a “1” into the file to turn the LED on and a “0” to turn it off.

## led Triggering

### Setting up a trigger ties an LED to a kernel event

```
# cat /sys/devices/platform/leds-gpio/leds/am335x\:EVM_SK\:heartbeat/trigger  
none nand-disk mmc0 mmc1 timer [heartbeat] rfkill0 phy0rx phy0tx phy0assoc phy0radio
```

Brackets around [heartbeat] indicate this LED is configured to trigger on the system heartbeat event.

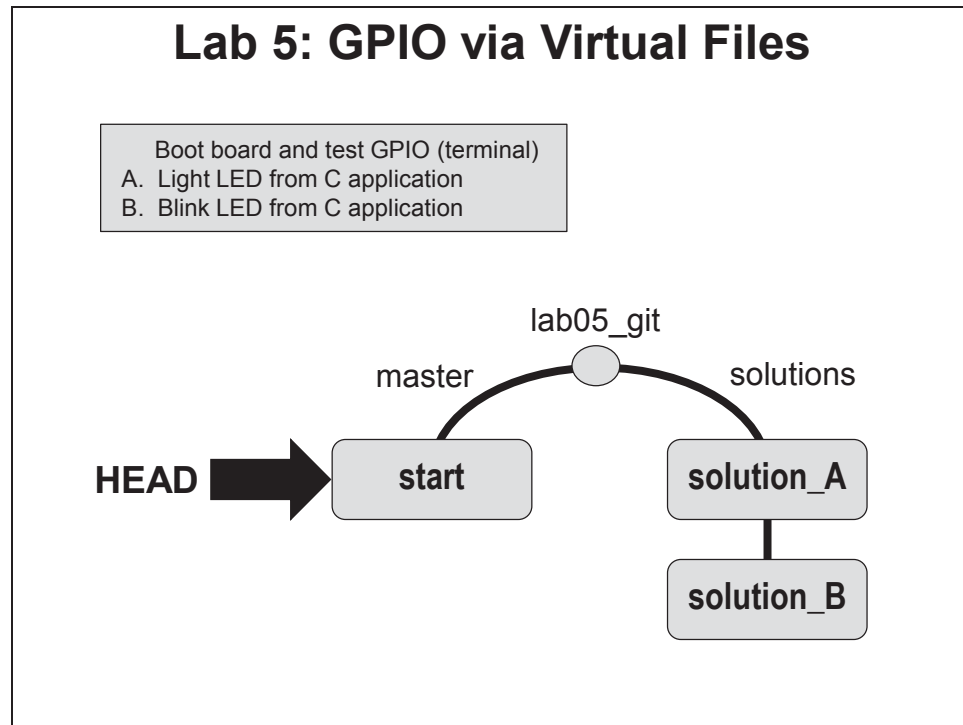
### To disable the heartbeat led

```
# echo none > /sys/devices/platform/leds-gpio/leds/am335x\:EVM_SK\:heartbeat/trigger
```

The leds-gpio driver also supports triggering of the LED from various system events, for instance to blink the led when NAND or the MMC/SD card interfaces or accessed. These system events are displayed when one prints the contents of the “trigger” virtual file corresponding to the given LED using the “cat” command from the terminal. The brackets (“[ ]”) surrounding one of the triggers indicates the current trigger setting.

The triggering is set by writing the desired trigger as a string into the “trigger” virtual file. Note that one of the triggers is “none” which will disable triggering of the led, as shown.

## Lab 5: GPIO via Virtual Files





# Module 06: Linux Drivers

---

## Introduction

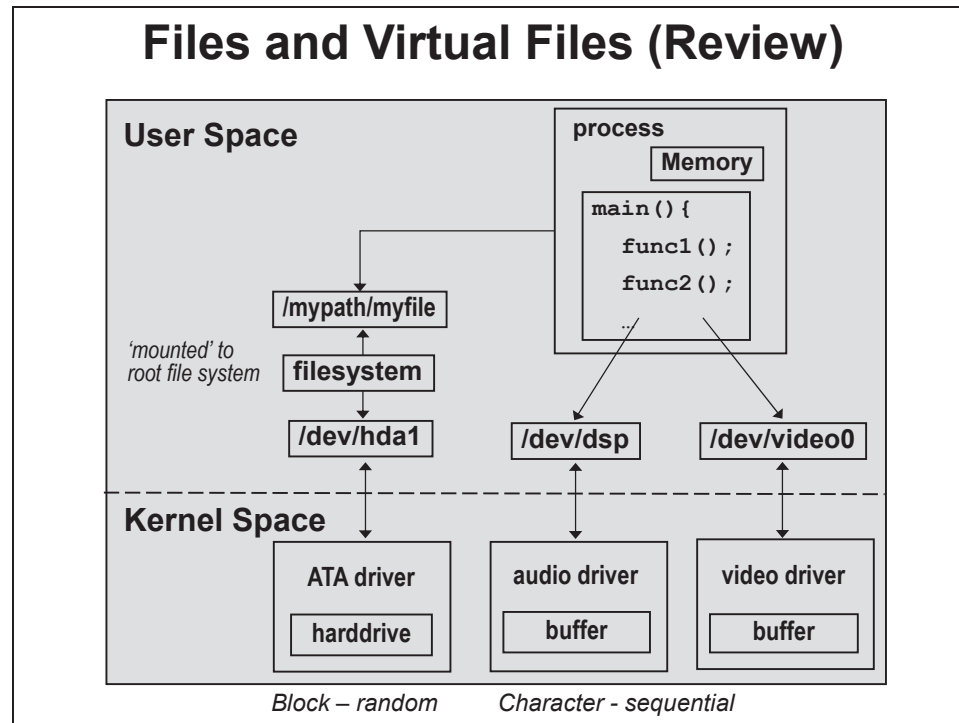
A basic feature of nearly any modern operating system is the abstraction of peripherals via device drivers. Linux provides a basic open-read-write-close interface to peripheral drivers that is a close parallel to the file manipulation interface described in the previous module.

This module provides a brief discussion on the differences between the basic driver model and the file I/O model then delves into two case studies: the UART driver and the audio driver (ALSA). The module wraps up with a related side topic: Linux signal handling.

## Module Topics

|   |             |
|---|-------------|
| <b>Module 06: Linux Drivers .....</b>                 | <b>6-1</b>  |
| <i>Module Topics.....</i>                             | <i>6-2</i>  |
| <i>Driver Basics .....</i>                            | <i>6-3</i>  |
| <i>UART Driver.....</i>                               | <i>6-7</i>  |
| <i>Advanced Linux Sound Architecture (ALSA) .....</i> | <i>6-11</i> |
| <i>Signal Handling.....</i>                           | <i>6-17</i> |
| <i>Lab 6: UART Driver .....</i>                       | <i>6-20</i> |

## Driver Basics



The basic driver model was discussed in the previous module. That module focused primarily on block devices, which use filesystems to manage files. This module will examine the character drivers that are used to access data streams, such as UART drivers, audio drivers or video drivers.

## Accessing Files (Review)

Manipulating files from within user programs is as simple as...

The diagram illustrates the use of standard C file I/O functions. It features a central code block with four lines of code. Arrows point from descriptive labels to specific parts of the code: 'File descriptor / handle' points to 'myFileFd' in the first line; 'Directory previously mounted' points to '/mnt/harddrive/' in the first line; 'File to open...' points to 'myfile' in the first line; 'Permissions' points to 'rw' in the first line; 'Array to read into / write from' points to 'aMyBuf' in the second and third lines; 'size of item' points to 'sizeof(int)' in the second and third lines; '# of items' points to 'len' in the second and third lines; and 'File descriptor / handle' points to 'myFileFd' in the second and third lines.

```
myFileFd = fopen("/mnt/harddrive/myfile", "rw");  
fread ( aMyBuf, sizeof(int), len, myFileFd );  
fwrite( aMyBuf, sizeof(int), len, myFileFd );  
fclose( myFileFd );
```

Additionally, use `fprintf` and `fscanf` for more feature-rich file read and write capability

When accessing files on the filesystem of a block device, the “`fopen`,” “`fread`,” “`fwrite`” and “`fclose`” functions are typically employed.

## Using Character Device Drivers

Simple drivers use the same format as files:

```
soundFd = open("/dev/dsp", O_RDWR);
read ( soundFd, aMyBuf, len );
write( soundFd, aMyBuf, len );
close( soundFd );
```

Additionally, drivers use I/O control (ioctl) commands to set driver characteristics

```
ioctl( soundFd, SNDCTL_DSP_SETFMT, &format);
```

### Notes:

- ◆ len field is always in bytes
- ◆ More complex drivers, such as V4L2 and FBDEV video drivers, have special requirements and typically use ioctl commands to perform reads and writes

When accessing character devices, the non-“f” version of these functions, “open,” “read,” “write” and “close” are typically used instead. These functions are very similar, although instead of using file pointers, these functions use a file descriptor. File descriptors are just “int” (integer) types. The open call returns a unique integer for each device node opened, beginning with “3.” File descriptors 0-2 are reserved for the standard input stream, the standard output stream and the standard error stream.

Character drivers also introduce a new function, “ioctl” which stands for Input/Output Control. This function is used to configure the driver settings for such parameters as sample rate, sample format, number of channels, etc. Of course the driver settings are different for each type of driver – a video driver does not use the same parameters as a UART. Each driver supports a set of ioctl commands (the second parameter to the ioctl function is the command) and the driver documentation should be referenced to determine what commands are supported. The final parameter to ioctl is a void pointer, which allows the user to pass any type of data or structure via the pointer. The data structure expected depends on the command issued.

## Dynamic Driver Example – USB

```
beaglebone$ lsmod  
g_mass_storage  
beaglebone$ rmmod g_mass_storage  
beaglebone$ modprobe g_ether
```

**connect\_ethernet.sh**

**The connect\_ethernet.sh script manually removes the mass storage USB gadget driver from the Beaglebone's running kernel and then dynamically inserts the Ethernet-over-USB gadget driver.**

Drivers, just as any kernel module, may be built as dynamically loadable modules. This slide shows an example of manipulating kernel modules. First the “lsmod” (LiSt MODule) command is used to list the dynamic modules that are currently loaded into the Linux kernel. In this case there is only one, “g\_mass\_storage,” which is the mass storage USB gadget driver.

The user might then remove this mass storage driver in order to use the USB channel it occupies for another purpose. This is accomplished with the “rmmod” (ReMove MODule) command. Finally, the desired driver is loaded using modprobe, the “g\_ether” module, which is the Ethernet-over-USB gadget driver.

## UART Driver

### TTY Teletype Terminal



(Left) Early TTYs printed to paper.

(Right) screens were later added



**Name “teletype terminal” derives from historical hardware device that is rarely used today**

**All Linux textual terminals (shell, uart, modem, etc.) use the TTY standard**

**Basic `open`, `read`, `write` and `close` calls to use**

**Common practice is to use the `<termios.h>` library in place of direct `ioctl` control calls.**

The modern Linux UART driver is an evolution of the original teletype terminal driver. Teletype terminals were hardware developed in the 80's for users to communicate with a remote server. This is the reason that the names of device nodes for UART drivers typically begin with “tty.”

It is possible to control a UART driver with `ioctl` calls, but in practice, most programmers utilize the “termios” library, which provides simple functions for setting common UART parameters.

## Serial Port Configuration (Part 1)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <termios.h>

// Simple function to configure a serial port to 115200N1
void configure_serial(int fd)
{
    struct termios terminal_setting;

    tcgetattr(fd, &terminal_setting);

    // Set baud rate
    // Most drivers use input rate for both
    cfsetispeed(&terminal_setting, B115200);
    cfsetospeed(&terminal_setting, B115200);
```

This slide shows the basic procedure for configuring a UART using the termios library. First, a structure of the “termios” type is declared. In this example it is named “terminal\_setting.”

Next the “tcgetattr” function is used to fill the structure with the current UART settings. This is generally better practice than zero-ing out the fields of the structure as an initialization. By initializing the structure with the current settings, any parameter that is not modified will be left unchanged as opposed to set to whatever value happens to correspond to “0” or, even worse, to an uninitialized value.

Next the “cfsetispeed” and “cfsetospeed” functions are used to set the input and output speeds, respectively. Note that while the driver allows specification of separate speeds for input and output, not all hardware supports asynchronous input and output, so it is better for portability to set these to the same value unless there is a specific reason for having different input and output speeds on the UART.



## Serial Port Configuration (Part 2)

```
// clear number of bits field
terminal_setting.c_cflag &= ~CSIZE;
// set 8 bits
terminal_setting.c_cflag |= CS8;

// clear parity bit in flag (no parity)
terminal_setting.c_cflag &= ~PARENB;

// clear the CSTOPB flag in the mask
// CSTOPB = 1 : 2 stop bits
// CSTOPB = 0 : 1 stop bit
terminal_setting.c_cflag &= ~CSTOPB;

// send changes to terminal settings structure to kernel
// TCSANOW flag forces immediate change
tcsetattr(fd, TCSANOW, &terminal_setting);
}
```

The next three UART configuration settings are managed via a virtual register in the “terminal\_setting” structure that is called “c\_cflag.” The appropriate bit fields are configured in this virtual register using set and clear operations. By using the bitwise and operation (&=) with the inverse (~) of a bitfield such as CSIZE, the corresponding bits are cleared. The bitwise or operation (|=) may then be used to set the bitfield with the desired value.

The “number of bits” field contains more than one bit, so it must be cleared and then set. The parity and stop bits are single-bit flags, and therefore only require a clear or set operation as desired.

After the terminal\_setting structure has been populated, the “tcsetattr” function is used to configure the UART hardware using its settings.

## Serial Driver “Hello World”

```
int main( void )
{
    char hello[40]="hello world\n";
    int fd;

    // Open serial device
    fd = open("/dev/ttyO0", O_RDWR);

    configure_serial(fd);

    // Write a string (char array) to device
    write( fd, hello, strlen(hello));

    // Close fd to allow other programs access
    close(fd);

    return 0;
}
```

After configuring the UART, capturing and sending data is fairly simple and uses a straightforward implementation of “open,” “read,” “write” and “close” functions. First, the program opens the device node corresponding to the appropriate UART. Next, the write call is used to write a character string to the device node. This character string will be sent over the UART connection. The close call can be used to disconnect.

## Advanced Linux Sound Architecture (ALSA)

### ALSA Applications

- ◆ The ALSA driver provides 3 Linux applications to exercise the driver.
- ◆ While not fancy, record/play app's are useful for testing. The mixer is useful for choosing inputs/outputs.
- ◆ The three app's are:

|                |  |
|----------------|--|
| <b>arecord</b> | Record audio from an ALSA device to a file             |
| <b>aplay</b>   | Play recorded audio over an ALSA device                |
| <b>amixer</b>  | Select input sources and adjust relative volume levels |

The most widely used Linux sound driver currently is the “Advanced Linux Sound Architecture” (ALSA) driver. In addition to the driver itself, the maintainers of this driver also provide three sample applications – “arecord,” “aplay” and “amixer.” In addition to being useful applications in their own right, they are excellent for testing the also driver on a new system.

For instance, developers who create their own unique hardware may need to port the ALSA driver for sound support on their system. Using the provide ALSA applications for test is recommended because they are known-good applications and because they provide quite a bit of useful debugging feedback.

## ALSA Library API

|   |                           |
|---|---------------------------|
| <b>Information Interface</b>  | <u>/proc/asound</u>       |
| <i>Status and settings for ALSA driver.</i>                                   |                           |
| <b>Control Interface</b>  | <u>/dev/snd/controlCX</u> |
| <i>Control hardware of system (e.g. adc, dac).</i>                            |                           |
| <b>Mixer Interface</b>  | <u>/dev/snd/mixer</u>     |
| <i>Controls volume and routing of on systems with multiple lines.</i>         |                           |
| <b>PCM Interface</b>  | <u>/dev/snd/pcmCXDX</u>   |
| <i>Manages digital audio capture and playback; most commonly used.</i>        |                           |
| <b>Raw MIDI Interface*</b>  | <u>/dev/snd/midiCXDX</u>  |
| <i>Raw support for MIDI interfaces; user responsible for protocol/timing.</i> |                           |
| <b>Sequencer Interface*</b>   | <u>/dev/snd/seq</u>       |
| <i>Higher-level interface for MIDI programming.</i>                           |                           |
| <b>Timer Interface</b>  | <u>/dev/snd/timer</u>     |
| <i>Timing hardware used for synchronizing sound events.</i>                   |                           |

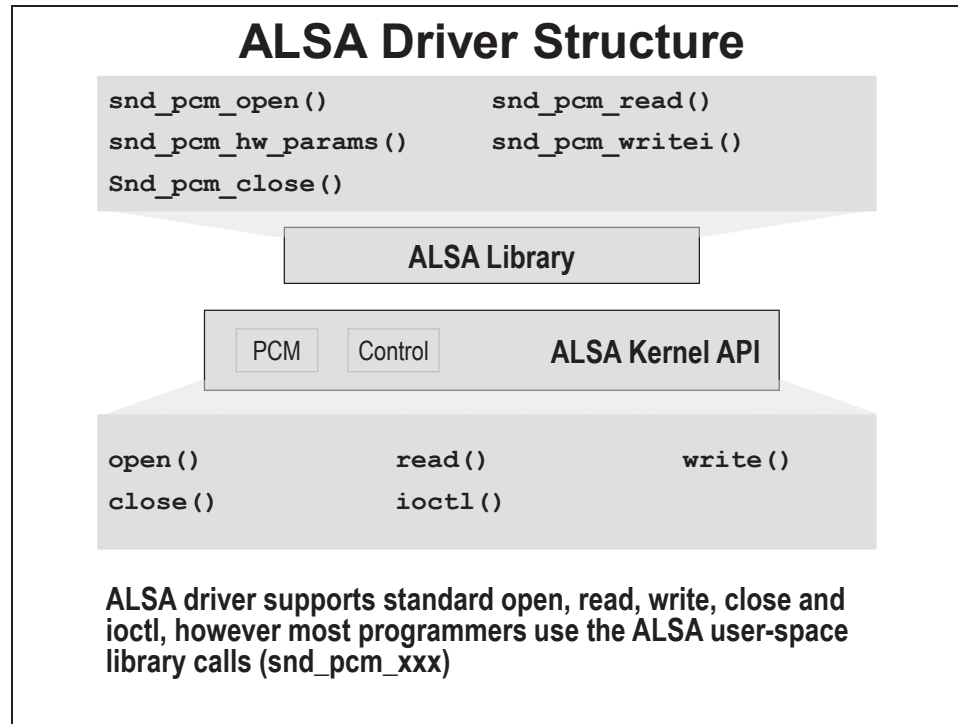
\* Not implemented in current TI provided driver.

The standard ALSA driver uses quite a number of device nodes. This is due to the fact that the driver supports an entire sound system, connected by a mixer. This sound system can support multiple sound cards, each having multiple devices, i.e. capture and playback channels. Support for multiple sound cards is more common in a desktop world than in embedded devices, but nonetheless, the Arago distribution uses ALSA's defined device node structure for parallelism and portability.

There is a control node for each sound card at “/dev/snd/controlCX” where the integer “X” corresponds to the card number. And for each device, i.e. capture or output channel, on the card, there is a PCM interface node, which is the data channel, at “/dev/snd/pcmCXDX,” where “CXDX” specifies the card number and the device number on the card.

There is also a mixer interface for setting volume levels between the channels and even mixing them together, as well as a timer interface for collecting timestamps.

The good news is that, while all of these device nodes may be intimidating to the new user, most programmers use something called the “ALSA userspace library,” which is what will be studied in this module, and this library has simplified function calls that do not require the user to interface directly to this myriad of control nodes.



While programmers are certainly able to access the ALSA device nodes listed on the previous slide using the standard “open,” “read,” “write,” “close” and “ioctl” commands, most users prefer the ALSA userspace library.

The ALSA userspace library simplifies interaction with the ALSA driver using similar calls, i.e. “snd\_pcm\_open,” which is analogous to the standard “open.” These userspace library calls are the interface that is presented in this module.

## Opening and Closing Devices

### Devices:

**hw:i,j**        **i = card number, j = device number**

**default**       **alias for hw:0,0**

```
/* Open the device */
rc = sound_pcm_open( &handle, "default",
                    SND_PCM_STREAM_PLAYBACK, 0 );

/* Close the device */
/* Drain call blocks until last data plays */
snd_pcm_drain(handle);
snd_pcm_close(handle);
```

Opening an ALSA device may be accomplished using the “snd\_pcm\_open” function. The first parameter to “snd\_pcm\_open” is an ALSA handle, which is passed by address. The handle is passed by address so that the function is able to overwrite its value. The handle is typically initialized to NULL upon entering the “snd\_pcm\_open” function, and the function overwrites the handle with a valid value.

The second field specifies the ALSA device to open. Hardware devices are given the format “hw:i,j” where “i” is the card number and “j” is the device number. Since systems that use the ALSA driver should have at least one channel, there is an alias supported named “default.” This alias is equivalent to “hw:0,0”

The device may be closed when it is no longer in use by using the “snd\_pcm\_close” call. Though not required, it is recommended to also use the “snd\_pcm\_drain” call before calling close. This is a blocking call that will not return until all backlogged samples to the driver have been played over the audio port. This eliminates the possibility of clipping the final samples of the output buffer before they are played when closing the device.

## ALSA Configuration Example

```
/* Allocate a hardware parameters object. */
snd_pcm_hw_params_allocate(&params);
/* Fill it in with default values. */
snd_pcm_hw_params_any(handle, params);
/* Set the desired hardware parameters. */
/* Interleaved mode */
snd_pcm_hw_params_set_access(handle, params,
                             SND_PCM_ACCESS_RW_INTERLEAVED);
/* Signed 16-bit little-endian format */
snd_pcm_hw_params_set_format(handle, params,
                              SND_PCM_FORMAT_S16_LE);
/* Two channels (stereo) */
snd_pcm_hw_params_set_channels(handle, params, 2);
/* 44100 bits/second sampling rate (CD quality) */
val = 44100;
snd_pcm_hw_params_set_rate_near(handle, params, &val, &dir);
/* Write the parameters to the driver */
rc = snd_pcm_hw_params(handle, params);
```

This slide shows the typical configuration procedure for an ALSA device. While there are a lot of function calls shown, each call is fairly simple to understand, and if viewed as “cut and paste” code, this configuration procedure is not too difficult to use.

The “`snd_pcm_hw_params_allocate`” call is used to allocate a parameter structure. The `params` pointer is passed by address, allowing the NULL-initialized pointer to be overwritten with the location of the allocated structure.

The “`snd_pcm_hw_params_any`” function fills the parameter structure with default values and then “`snd_pcm_hw_params_set_access`” sets the parameters to use interleaved left and right channels.

The “`snd_pcm_hw_params_set_format`” call sets the data format to signed 16-bit little endian, and the “`snd_pcm_hw_params_set_channels`” sets the number of output channels to 2, or stereo mode.

The last configuration function is a little trickier. The “`snd_pcm_hw_params_set_rate_near`” function passes the sample rate by address. This allows the sample rate to be overwritten. This parameter is used as both an input and an output value to the function. This is because most audio hardware supports only discrete values of sample rate. The set rate near function sets the sample rate to the value that is closest to the supplied value and, if this is not the supplied value, overwrites the variable with the sample rate that was actually set. In this example, 44,100 Hz is supported, so this value would be set. If the user instead specified 44,090 Hz, this value would be set to 44,100 Hz, the nearest supported value, and the variable would be overwritten with 44,100.

The final call takes the parameters structure, and uses it to set the values in the hardware. Note that all of the preceding calls only modify the parameter structure, and it is not until the final call that the hardware is updated.

## ALSA Read/Write Example

```
snd_pcm_hw_params_get_period_size(params, &frames, &dir);
size = frames * 4;
buffer = (char *) malloc(size);
while (loops-- > 0) {
    rc = read(0, buffer, size);
    if (rc < size) {
        fprintf(stderr, "end of file on input\n");
        break;
    }
    rc = snd_pcm_writei(handle, buffer, frames);
    if (rc < 0)
        fprintf(stderr, "error from writei: %s\n",
                snd_strerror(rc));
}
```

Once the ALSA driver has been configured, data is read from the driver using “snd\_pcm\_read” or output to the driver using “snd\_pcm\_writei” (interleaved channels) or “snd\_pcm\_write” (non-interleaved channels.)

Data is read or written in frames. The “snd\_pcm\_hw\_params\_get\_period\_size” function is used to determine the native or internal buffer size used by the driver, which is provided in number of frames. When operating in interleaved mode, a frame is a single sample from each channel. In this example there are two channels, and each channel uses 16-bit data, so a frame is 4 bytes. The program converts the number of frames per period, i.e. per each internal buffer, into the number of bytes per period, and then enters a loop to read a full period of data and pass to the driver.

It is not required to pass data to the driver according to its native period, i.e. to pass a full block of data corresponding to one internal buffer. It is only required to pass an integer number of frames. However, passing data according to the internal frame size of the driver is more efficient, and so this is what is shown in the example.



# Signal Handling

## Linux Signals

**A signal is an event generated by Linux in response to some condition, which may cause a process to take some action when it receives the signal.**

- ◆ **“Raise” indicates the generation of a signal**
  - ◆ Memory segment violations
  - ◆ Floating-point processor errors
  - ◆ Illegal instructions
  - ◆ User generated at terminal
  - ◆ Send from one process (program) to another
  
- ◆ **“Catch” indicates the receipt of a signal**
  - ◆ Application programmers may register signal handling routines
  - ◆ If no user signal handler is registered, a default signal handler is used

Linux manages a set of signals that may be generated by the Linux kernel itself, by an application, or by user input from a terminal. The generation of a signal is termed “raising” the signal. These signals are sent to a running application, i.e. Linux process, which is said to “catch” the signal.

The Linux kernel raises signals that are caught by the application that is running when the signal is raised. Signals generated by terminal input are sent to whichever application is running in the foreground of that terminal, and signals that are generated from an application may be sent to any process via the process id.

When a process “catches” a signal, it executes a callback function that is registered as the signal handler for that signal. Linux registers a default set of signal handlers for each newly generated process. Users may also substitute their own signal handlers in place of the defaults.

## Signals defined in signal.h

| Signal  | Value | Action | Comment   |
|---------|-------|--------|---|
| SIGHUP  | 1     | Term   | Hangup detected on controlling terminal or death of controlling process |
| SIGINT  | 2     | Term   | Interrupt from keyboard   |
| SIGQUIT | 3     | Core   | Quit from keyboard  |
| SIGILL  | 4     | Core   | Illegal Instruction   |
| SIGABRT | 6     | Core   | Abort signal from abort(3)  |
| SIGFPE  | 8     | Core   | Floating point exception  |
| SIGKILL | 9     | Term   | Kill signal   |

*\* Note, this is not a complete list*

This slide lists some of the most commonly used Linux signals and the integer value assigned to each. For instance, “SIGINT,” which is enumerated as signal 2, corresponds to ctrl-c from the terminal.

Note that SIGKILL is a special signal. It is the only signal which cannot be caught by a signal handler. Instead, the Linux kernel will always catch SIGKILL and terminate the running program.

## Signal Registration Example

```
int main(int argc, char *argv[])
{
    int status = EXIT_SUCCESS;
    void *audioThreadReturn;

    /* Set the signal callback for Ctrl-C */
    signal(SIGINT, signal_handler);

    /* Call audio thread function */
    audioThreadReturn = audio_thread_fxn((void *) &audio_env);

    if( audioThreadReturn == AUDIO_THREAD_FAILURE ){
        printf("audio thread exited with FAILURE status\n");
        status = EXIT_FAILURE;
    } else
        printf("audio thread exited with SUCCESS status\n");

    exit(status);
}

/* Callback called when SIGINT is sent to the process (Ctrl-C). */
void signal_handler(int sig)
{
    DBG("Ctrl-C pressed, cleaning up and exiting..\n");
    audio_env.quit = 1; // used as while loop condition
}
```

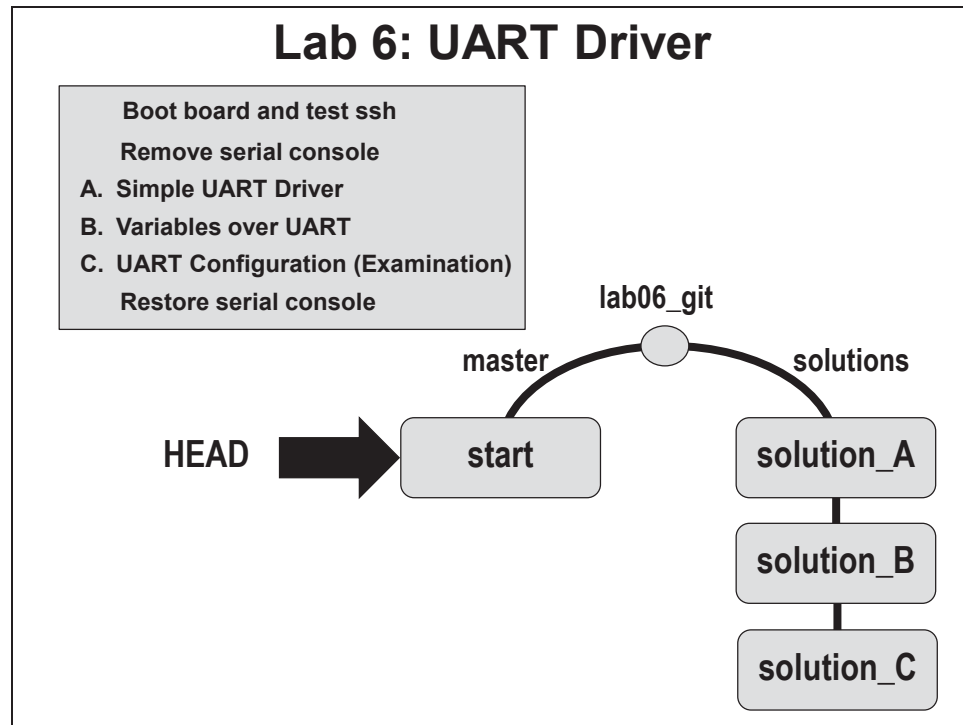
This code demonstrates the process for registering a signal handler.

A signal handler is just a function with a required prototype. In order to be registered as a signal handler, a function must take a single integer value and return nothing. The integer value that is passed into the signal handler is the enumerated value associated with the signal, for instance “2” for the “SIGINT” signal as shown on the previous slide.

The function is registered as a signal handler using the “signal” function call. The first parameter to the signal function is the signal to register the signal handler for. In the above example, the code is registering a signal handler for the “SIGINT” signal.

The second parameter is the function pointer of the signal handler, which in C and C++ is the name of the function. Recall that when the signal handler is called, Linux will pass the enumerated value of the signal that was generated via the integer parameter of the signal handler function. In this code, the “signal\_handler” function is registered only to a single signal, SIGINT, so this field is ignored. In some systems, the same signal handler is used to handle multiple signals. This is accomplished via successive calls of the “signal” function. If the same signal handler is used to catch multiple signals, then the signal value passed into the signal handler may be required to determine the proper response.

## Lab 6: UART Driver



# Module 07: Linux Scheduler

---

## Introduction

Another common feature of nearly any operating system is the ability to run multiple tasks concurrently, often referred to as “multi-threading.” The Linux operating system has two primary features used for concurrent tasks. The first is a memory management feature via the Memory Management Unit (MMU) in which separate memory spaces are created for each process, i.e. application.

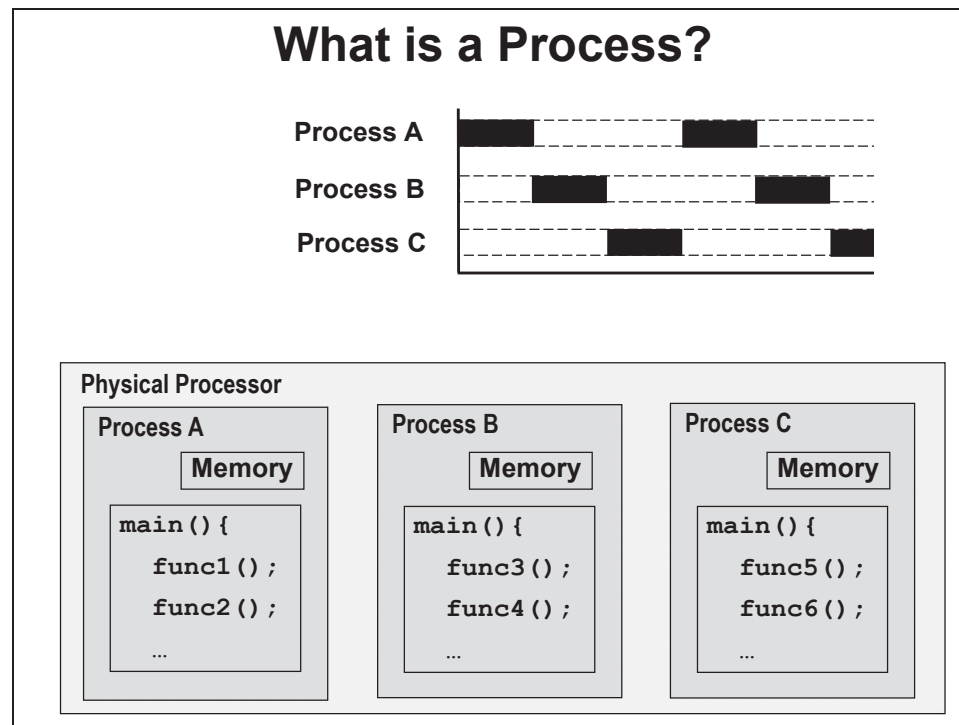
The second feature is the sharing of CPU cycles across different applications via separation into POSIX threads. The Linux scheduler also manages the context switching between different threads of execution, including saving and restoring the system stack, saving and restoring CPU registers and reprogramming of the memory management unit if necessary.

This module begins with an overview of processes and threads, then discusses thread synchronization using semaphore objects. The module wraps up with a discussion of the usage of the two primary scheduling methodologies for threads: realtime and timeslicing.

## Module Topics

|   |             |
|---|-------------|
| <b>Module 07: Linux Scheduler .....</b> | <b>7-1</b>  |
| <i>Module Topics.....</i>               | <i>7-2</i>  |
| <i>Linux Processes .....</i>            | <i>7-3</i>  |
| <i>Linux Threads .....</i>              | <i>7-9</i>  |
| <i>Thread Synchronization .....</i>     | <i>7-13</i> |
| <i>Using Real-Time Threads .....</i>    | <i>7-19</i> |
| <i>Lab 7: Linux Scheduler.....</i>      | <i>7-28</i> |

# Linux Processes



A common requirement for many modern systems is the ability to run multiple applications concurrently using a single physical processor. This requires a scheduler to arbitrate resources between the applications.

The basic requirements for an application to run are program and data memory and CPU cycles. In some systems the program and data memory are assigned to each application statically using a linker. This has the advantage of simplicity, but it is less efficient since the memory has to be assigned to the application whether it is executing or not.

Linux uses dynamic, or run-time, linking to dynamically share memory between the running applications of the system. The memory spaces of each application are enforced by the Memory Management Unit (MMU) hardware, which is important for the robustness of the system. If a program mis-manages a pointer and attempts to access memory it does not own, the MMU will disallow the access and will generate a segmentation fault signal. This mechanism protects the Linux kernel's memory space from applications as well as protecting applications from each other.

Applications also need to share CPU cycles. This arbitration is the primary function of the scheduler. The CPU can only execute one thread of code at a given moment, so the scheduler must use time-division multiplexing to share the CPU between applications over time. The timeline in the upper right-hand corner shows a typical sharing of CPU between executing threads.

The sum of resources assigned to a given application are referred to in Linux as a "Process." The Process, then, is the context in which the application executes, which is managed by the Linux scheduler.

## Scheduling Methodologies

### Time-Slicing Threads

Scheduler shares processor run time between all threads with greater time for higher priority

- ✓ No threads completely starve
- ✓ Corrects for non-"good citizen" threads
- ✗ Can't guarantee processor cycles even to highest priority threads.
- ✗ More context switching overhead

### Realtime Threads

Lower priority threads won't run unless higher priority threads block (i.e. pause)

- ✗ Requires "good citizen" threads
- ✗ Low priority threads may starve
- ✓ Lower priority threads never break high priority threads
- ✓ Lower context-switch overhead

- ◆ Time-sliced threads have a "niceness" value by which administrator may modify relative loading
- ◆ Linux dynamically modifies processes' time slice according to process behavior
- ◆ With realtime threads, the highest priority thread always runs until it blocks itself

Linux employs two methodologies to arbitrate CPU usage between applications. The first is a simple mechanism referred to as "time-slicing." With time-slicing, the scheduler splits the CPU usage between all applications by proportion and cycles through each application to allow it to execute its "slice" of code on the CPU. By default each application is given an equal proportion, but this proportion may be modified using the "niceness" value. (More to follow.) However; regardless of niceness value, each application is guaranteed to receive some proportion of CPU time each cycle through the list.

The second scheduling methodology is real-time. In a real-time scheduler, threads are prioritized using an integer priority setting. The scheduler allows the highest priority thread(s) to execute until they relinquish the CPU by signaling the scheduler with some form of blocking, such as a "sleep" call, or more advanced forms such as a semaphore, which will be discussed later in this module.

The primary benefit of real-time scheduling is that it allows the developer to guarantee that certain high-priority threads will be given as many CPU cycles as required in order to meet their real-time requirements. The primary drawback of real-time scheduling is that the low-priority threads may receive no CPU cycles at all, a situation referred to as "starvation," so that using real-time threads requires more extensive planning and characterization from a system level.



## The Usefulness of Processes

### Option 1: Audio and Video in a single Process

```
// audio_video.c
// handles audio and video in
// a single application

int main(int argc, char *argv[])
{
    while(condition == TRUE){
        callAudioFxn();
        callVideoFxn();
    }
}
```

### Option 2: Audio and Video in separate Processes

```
// audio.c, handles audio only

int main(int argc, char *argv[]) {
    while(condition == TRUE)
        callAudioFxn();
}
```

```
// video.c, handles video only

int main(int argc, char *argv[]) {
    while(condition == TRUE)
        callVideoFxn();
}
```

#### Splitting into two processes is helpful if:

1. audio and video occur at different rates
2. audio and video should be prioritized differently
3. multiple channels of audio or video might be required (modularity)
4. memory protection between audio and video is desired

One question that may arise is why it is necessary at all to split a system into multiple applications. For instance, while a system that requires audio and video may certainly manage audio and video as two separate applications, it is also possible to simply combine both requirements into one large application.

While both approaches are possible, there are definite advantages to separating a system into logical application blocks, i.e. Linux processes. One primary advantage is synchronization of the application with input/output devices. In the example above, there is audio code that is processing audio data at the native rate that the data is being capture and video code that is processing video data at its native capture rate. If both sections of code are combined into a single application, then the application author must calculate the exact ratio between sample rates and incorporate this into the program, as well as creating code to handle drift between the data sets over time. Ultimately, this application author is creating a small scheduler within their code. By separating the audio and video code into separate applications, each application is synchronized individually to the underlying data channel.

Another advantage to separating code across multiple processes is the ability to prioritize one application over another using real-time threads.

## Terminal Commands for Processes

|                                   |  |
|-----------------------------------|--|
| # <b>ps</b>                       | <b>Lists currently running user processes</b>                    |
| # <b>ps -e</b>                    | <b>Lists all processes</b>                                       |
| # <b>top</b>                      | <b>Ranks processes in order of CPU usage</b>                     |
| # <b>kill &lt;pid&gt;</b>         | <b>Ends a running process</b>                                    |
| # <b>renice +5 -p &lt;pid&gt;</b> | <b>Changes time-slice ranking of a process</b><br>(range +/- 20) |

Here are a few commonly used terminal commands that relate to Linux processes. The “ps” command is used to list the currently running processes within a given context. By adding the “-e” flag, all processes across the system are listed.

The “top” command functions much as “ps” with the “-e” flag, but this command also ranks the processes by CPU usage, updating every two or three seconds. This command is useful when the system is performing sluggishly and the administrator want to track down the culprit application.

Both the “ps” and “top” commands list an integer value for each process. This integer is referred to as the “process id” or pid for short. Each process has a unique pid. Once the pid has been determined for a given application process, the kill command can be used to terminate the application.

Also, the “renice” command can be used to set niceness values for processes. The default niceness for processes is 0. When an application has a positive niceness, it surrenders a portion of its CPU cycles back to the system to be shared across the other processes. When an application has a negative niceness, it steals CPU cycles from the rest of the system.

## Launching a Process – Terminal

```

user:~/workdir/bootcampstarter/lab_soln/lab6_soln - Shell - Konsole
Session Edit View Bookmarks Settings Help

root@92.168.10.20:/mnt/bootcamp/lab_soln/lab6_soln/release# ./lab6_soln &
[1] 979
root@92.168.10.20:/mnt/bootcamp/lab_soln/lab6_soln/release# /dev/fb/0 initiali
ed with resolution 720x480 and 16 bpp.

root@92.168.10.20:/mnt/bootcamp/lab_soln/lab6_soln/release# ps
  PID TTY          TIME CMD
  975 pts/0        00:00:00 bash
  979 pts/0        00:00:09 lab6_soln
  980 pts/0        00:00:00 ps

root@92.168.10.20:/mnt/bootcamp/lab_soln/lab6_soln/release# kill 979
root@92.168.10.20:/mnt/bootcamp/lab_soln/lab6_soln/release# ps
  PID TTY          TIME CMD
  975 pts/0        00:00:00 bash
  981 pts/0        00:00:00 ps

[1]+  Terminated                  ./lab6_soln
root@92.168.10.20:/mnt/bootcamp/lab_soln/lab6_soln/release#

```

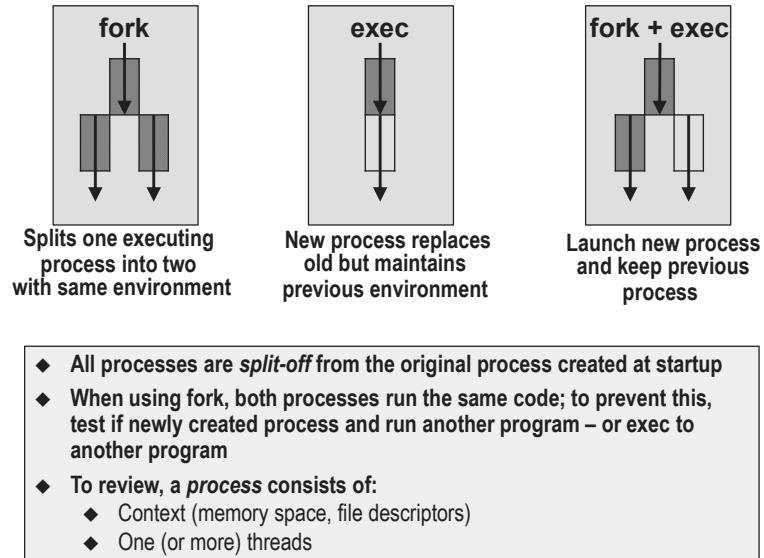
This terminal capture shows some basic process manipulation. The first line, surrounded by the red box, is used to launch an application as a process. The name of the application is preceded by “./” to indicate that the application to be executed exists in the current directory. (A single period is Linux terminal shorthand for current directory.) This is required because by default, the PATH variable in Linux does not contain the current directory.

The trailing ampersand (&) launches the application in the background of the terminal. Normally when an application is launched, the standard input stream to the application is attached to the Human Input Device (HID) associated with the terminal, usually a keyboard. Since the terminal input is attached to the application, no more terminal commands may be entered until the application exits. When an application is launched in the terminal background, the terminal input stream is detached from the application, allowing the user to type commands into the terminal.

After launching the application, the “ps” command is executed, as shown in the blue box. Three processes are shown by this command. The first process is the bash shell. This is the program that receives commands that are typed into the terminal and interprets them. The next process listed is the “lab6\_soln” application that was previously launched. Finally, the “ps” command itself is shown. Despite being short-lived, the ps command is launched as an application, and therefore has its own process id during the period in which it is executing.

Using the pid of the lab6\_soln application, 979, and the kill command, the lab6\_soln is terminated. Then a subsequent issue of the “ps” command shows only the bash shell and the ps command. Note that the second iteration of the ps command has a new process id. This is because the first issuing of ps launched as an application and terminated. The second time ps is run, it is launched as a new application and therefore is given a new process id.

## Side Topic – Creating New Processes in C



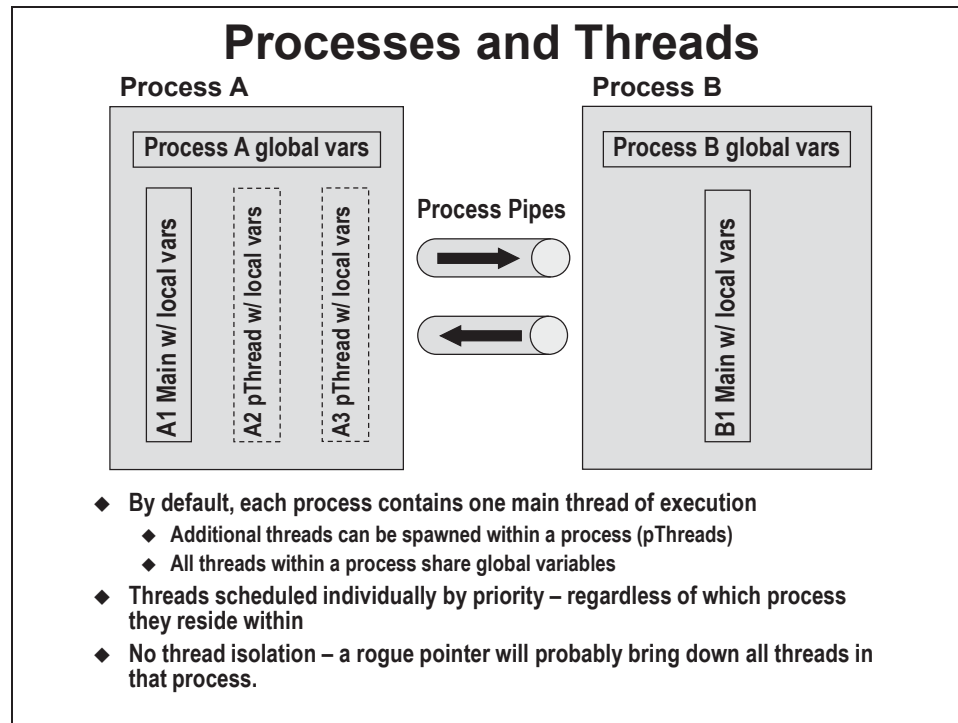
The previous slide shows how to launch an application from the Linux terminal. An application may also be launched from within another application using the system function calls of “fork” and “exec.”

“fork” is an interesting function. When “fork” is called within a running process Linux creates a nearly-identical copy of the calling process. The originating process is referred to as the “parent,” and the copy is referred to as the “child.” The implementation of Linux is such that all processes within the system are actually children of the init process, which is the first process created when the system boots.

So “fork” allows the user to create a new process, but it is always a copy of the parent process, so the new process is running the same application program as the original. In many cases the user wants to execute a different application than the application that is executing within the parent process. This may be accomplished via the “exec” function. The “exec” function will execute any command, i.e. application, that is specified as a parameter. However, the exec function executes the command within the context of the calling process. As a result, the “exec” call does not return until the command that it is executing has completed. This means that if “exec” is used to launch a media player that is used to view a two-hour movie, the exec call will not return for two hours.

Often what the user wants is to launch a new application, but launch it within the context of a new process. One way that this may be accomplished is with a combination of fork and exec. First, the fork call is made to create a child process. The parent process then continues with its original application, and the program within the child process uses the “exec” call to launch a new application.

# Linux Threads



When an application is launched, Linux places the application within the context of a Linux process, which defines a memory space, one or more execution threads and various system context such as file descriptors.

All processes begin with a single thread of execution, the main thread. This is the entry point to the application, and in C/C++ programs it corresponds to the “main” function. Linux also provides a mechanism for adding new execution threads within a process. Each new thread requires an entry point, and after multiple threads have been entered, the threads execute in parallel, their CPU usage arbitrated by the Linux scheduler.

Because the process defines a memory space and a set of file descriptors, all threads within the same process share global variables, file descriptors and any other process context. Threads that are in separate processes do not share global variables, however, as they exist in separate memory spaces. In order to communicate between processes, then, another mechanism must be employed. One of the most fundamental inter-process communication objects in Linux is called a process pipe. Process pipes are uni-directional message FIFO's. Processes can implement bi-directional communication using two process pipes.

## Threads vs Processes

|                   | Processes | Threads  |
|-------------------|-----------|----------|
| Memory protection | ✓         | ✗        |
| Ease of use       | ✓         | ✗        |
| Start-up cycles   | ✗         | ✓        |
| Context switch    | ✗         | ✓        |
| Shared globals    | no        | yes      |
| Scheduled entity  | no        | yes      |
| Environment       | program   | function |

This chart shows some of the fundamental differences between processes and threads. Because processes define a memory space, there is memory protection between processes, enforced in hardware by the Memory Management Unit. Threads within the same process do not have memory protection so one thread may potentially corrupt the memory space of another thread within its same process space.

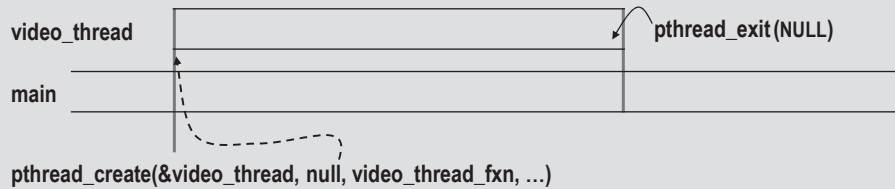
Processes are generally considered easier to use than threads because they provide more protection in the form of separation of resources, and because implementing multiple processes is as simple as writing and compiling multiple applications and then launching them. Threads are a little more difficult to use because care must be taken to avoid memory corruption across threads and and because a new set of API must be learned to launch new threads within a process.

One of the major advantages of threads as compared to processes is that they have lower overhead both in terms of startup cycles and context switch cycles. Also, threads within the same process are able to share global variables.

Note that threads are the entity that the Linux scheduler operates on. The Linux scheduler does not schedule processes, it schedules the main thread within a newly launched process and it schedules any additional threads that are spawned within the process. This means that real-time threads are always scheduled in order of their priority, regardless of which processes those threads exist within.

The environment of a process is a program. If a user compiles and launches an application, the application launches within the context of a new process. The environment of threads is a function. The entry point of the main or implicit thread of each application is the main function within the application, and to implement new threads within the application, the programmer will create new functions as entry points.

## pThread Functions – Create & Exit



```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
    void *(*start_routine)(void *), void *arg);
pthread_join(pthread_t thread, void **retval);
pthread_exit(void *retval);
```

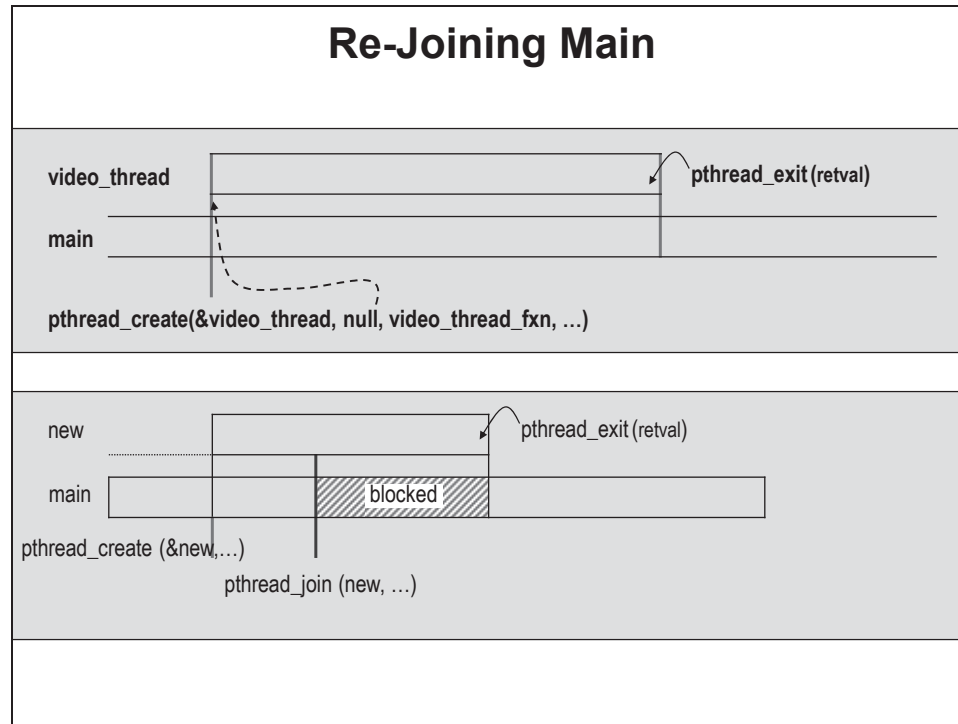
What if there's nothing for main() to do?

New threads are created within a process using the “pthread\_create” function. “pthread” is short for “POSIX Thread,” where POSIX is an old Unix and now Linux standard. The first parameter passed to “pthread\_create” is the address of a thread handle. The thread handle is passed by address because the handle used is typically initialized to NULL, but then its value is overwritten by the pthread\_create function call with the handle of the newly created thread.

The second argument to pthread\_create is an attribute structure. The third parameter is a function pointer. The function passed is the entry point of the new thread, and must have a specific prototype, which is that it has a single pointer argument and returns a single pointer. Because these are declared as “void \*” the pointers that are passed and returned may point to anything, allowing the user to define structures holding multiple arguments and return values and then pass the entire structures.

“pthread\_exit” is used to exit from the thread and allows the programmer to specify a pointer as a return value. As previously mentioned, this is a “void \*” so it may point to anything that the author wishes to return.

There is an issue, however, with returning a value from the newly created thread. The issue is that the return value is not valid until the thread has exited. How does the calling thread, in this case main(), know when the child thread has exited and therefore the return value is valid? This is the purpose of the “pthread\_join” function.



If the return value of the new thread, “video\_thread” in this example, is to be captured in main, then the main thread should call “pthread\_join” in order to capture the return value. The pthread\_join function is a blocking function. Blocking will be explored further in the next section, but in this case the important characteristic of “pthread\_join” is that when the function call is made in main, the main thread will not return from the “pthread\_join” function until the child thread exits.

The user should be wary when making this function call. If the thread that is joined to never exits, then the “pthread\_join” call will never return, locking up the main thread. Note that if “pthread\_join” is called after the child thread has already exited, there is no adverse effect. The “pthread\_join” call will immediately return, and the return value will still be available for the parent thread to capture.



# Thread Synchronization

## Thread Synchronization (Polling)

```
void *threadA(void *env){
    int test;
    while(1){
        while(test != TRUE) {
            test = (volatile int) env->driverComplete;
        }
        doSomething(env->bufferPtr);
    }
}
```

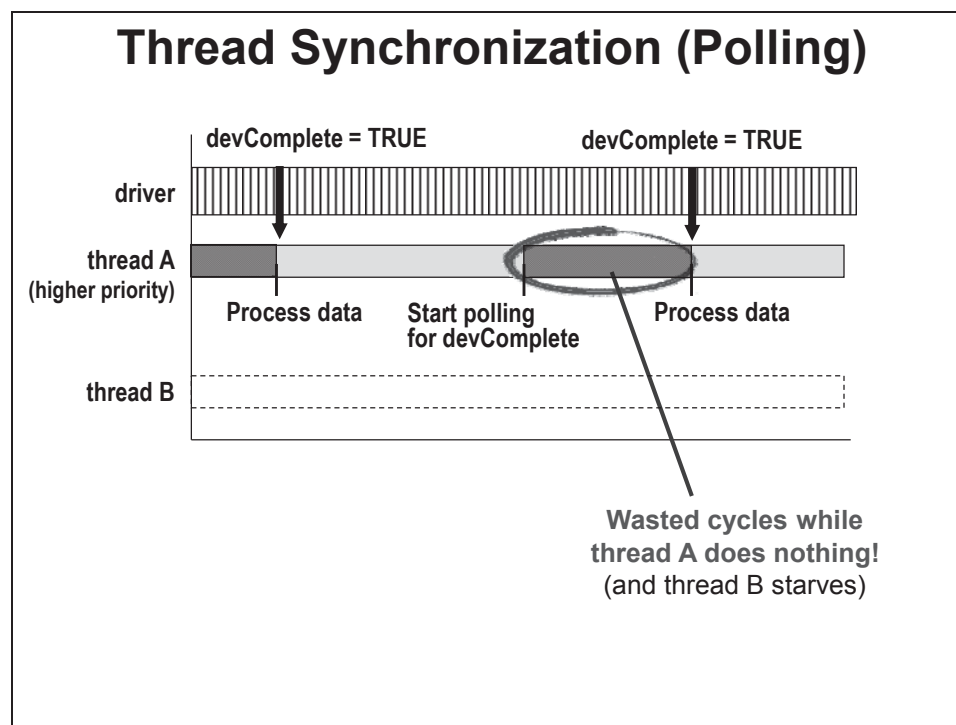
} Polling  
(spin loop)

- ◆ Thread A's doSomething( ) function should only run after the driver completes reading in a new buffer
- ◆ Polling can be used to halt the thread in a spin loop until the driverComplete flag is thrown.
- ◆ But polling is inefficient because it wastes CPU cycles while the thread does nothing.

Multithreaded systems often require a means of synchronizing the actions of various threads to each other. For instance, applications are often synchronized to driver channels that trigger the application according to the receipt of a new frame of data or the output of a previous frame of data.

There are various methods that may be employed to synchronize threads in a system. One of the most basic is called a polling loop, also known as a spin loop. This is not generally considered a good programming practice because spin loops waste CPU cycles, but they are also simple to understand and will be used here as a comparison point for the more complex technique of using semaphores.

A spin loop uses a global test variable as a means of communicating between two threads. In this example, the driver thread writes a “TRUE” value into the global variable when it captures a full frame of data. The application tests the global variable within a simple polling loop that polls the variable continuously until it transitions from “FALSE” to “TRUE” and then drops out of the polling loop to process the newly received frame of data.



A polling loop does successfully solve the problem of thread synchronization as shown on the above timeline. When the driver sets the `devComplete` global variable to “TRUE” it signals the application to drop out of its spin loop and process the data frame. The issue, however, occurs after the frame has been processed. After the frame is processed, the application drops into another spin loop to wait for the new frame of data to become available before it attempts to process it. The uses CPU cycles, which are basically wasted as the CPU performs the operation of testing the global variable over and over again.

This is particularly problematic when using real-time threads, because if the thread that implements a spin loop is a high-priority real-time thread it will lock out all of the lower priority threads in the system.

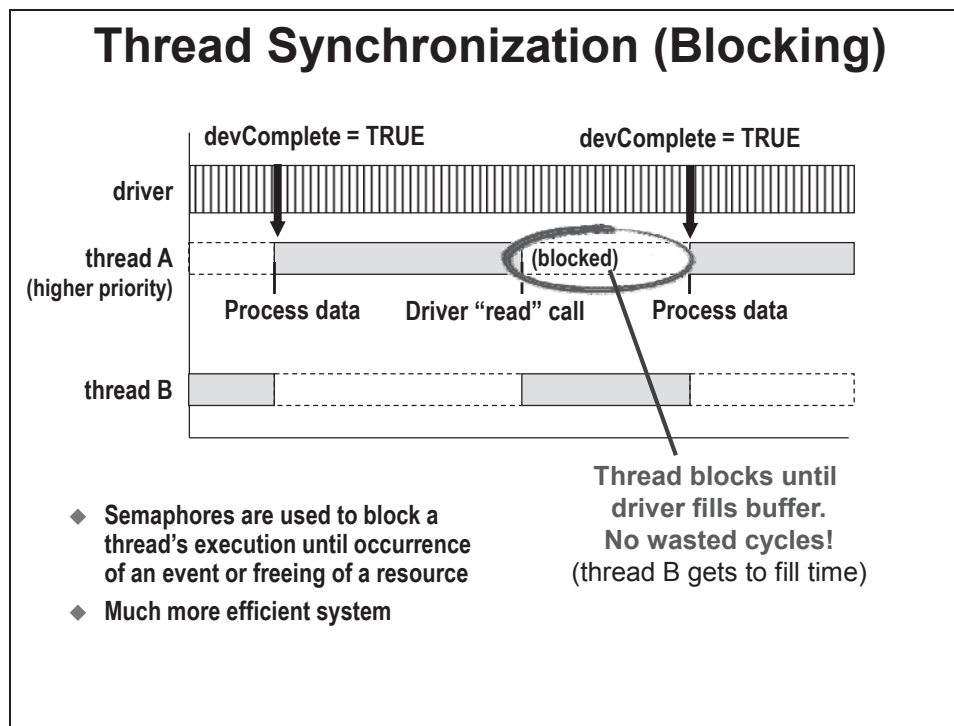
## Thread Synchronization (Blocking)

```
void *threadA(void *env){  
while(1){  
    read(env->audioFd, env->bufferPtr, env->bufsize);  
    doSomethingNext(env->bufferPtr);  
}  
}
```

**Blocking**  
(waits till complete)

- ◆ Instead of polling on a flag, the thread blocks execution as a result of the driver's read call
- ◆ More efficient than polling because thread A doesn't waste cycles waiting on the driver to fill the buffer

An alternative method to polling loops is to use blocking function calls. Linux supports a number of blocking function calls. For instance, the “read” function that was introduced in the driver chapter is a blocking function. When the application author calls “read” to read a block of data from a driver, the “read” call blocks until the amount of data requested is available. When a function call blocks, the application does not return from the function call, so that any function calls which occur after the blocking function are not executed until the blocking function unblocks. This synchronizes the application to the blocking event, in this case the read of the specified number of samples.



The advantage of blocking calls versus polling loops is shown here. Just as with the polling loop, the blocking call synchronizes the application to the receipt of the new frame of data from the driver. However, while the application thread is blocked, another thread can run in its place. This is as opposed to a polling loop which uses CPU cycles to poll and therefore locks out lower priority threads.

## Semaphores

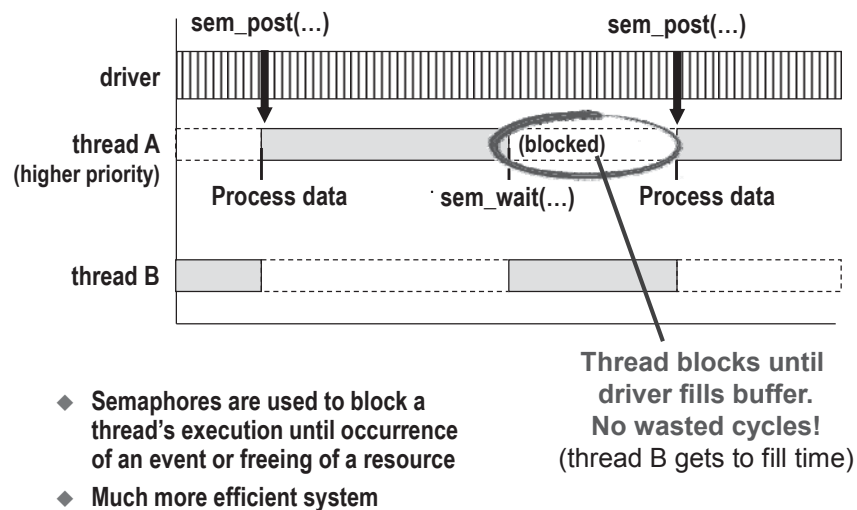
```
#include <semaphore.h>
void *threadA(void *env){
    sem_t mySem;
    sem_init(&mySem, 1, 0); // initial value of zero
    while(1){
        sem_wait(&mySem);
        doSomethingNext(env->bufferPtr);
    }
}
```

**Blocking**  
(waits till  
complete)

- ◆ A semaphore is the underlying mechanism of the read call that causes it to block

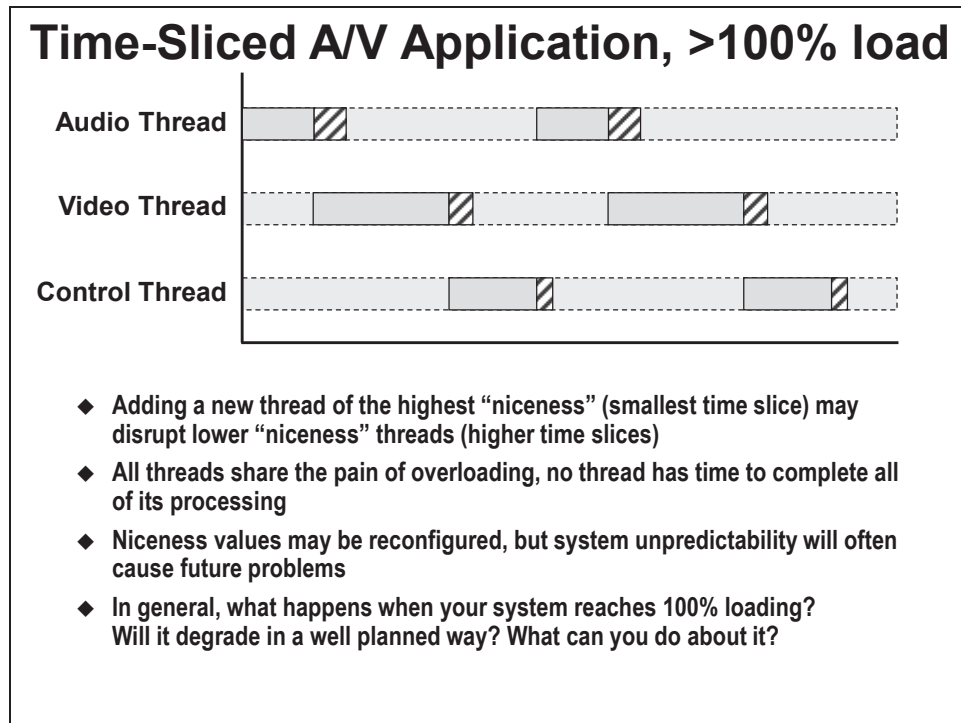
The fundamental Linux object that is used for blocking is called a semaphore. Linux functions that block, such as read, are implemented by calling “sem\_wait” within their subroutine. When sem\_wait is called, the calling thread blocks until an external thread calls “sem\_post.”

## Thread Synchronization (Semaphore)



In this example, the application calls `sem_wait` to block itself until a new frame of data is available. The driver then calls “`sem_post`” when a full frame of data has been captured, allowing the application thread to unblock and consume the frame of data. As before, while thread A is blocked, another thread, even a lower priority thread, is able to execute in its place.

## Using Real-Time Threads



To evaluate the benefits and negatives of time-slicing and real-time scheduling methodologies, we will evaluate a theoretical system. This system has an audio thread, a video thread and a control thread. Furthermore, we will consider the system when it is 125% loaded, meaning that the CPU is only capable of handling 80% of the system requirements in real time.

First we will examine a system based only on time-slice threads. In such a system the overloading is evenly dispersed across the three threads so that each thread completes approximately 80% of its requirements. In the diagram above, the solid bar indicates the 80% of the processing that is completed and the striped bar indicates the 20% of the processing that is not completed.

## Time-Sliced A/V Application Analysis

Audio Thread 

Audio thread completes 80% of samples

Video Thread 

Video thread drops 6 of 30 frames

Control Thread 

User response delayed 1mS

**All threads suffer, but not equally:**

- ◆ Audio thread real-time failure is highly perceptible
- ◆ Video thread failure is slightly perceptible
- ◆ Control thread failure is not remotely perceptible

**Note:**

Time-slicing may also cause real-time failure in systems that are <100% loaded due to increased thread latency

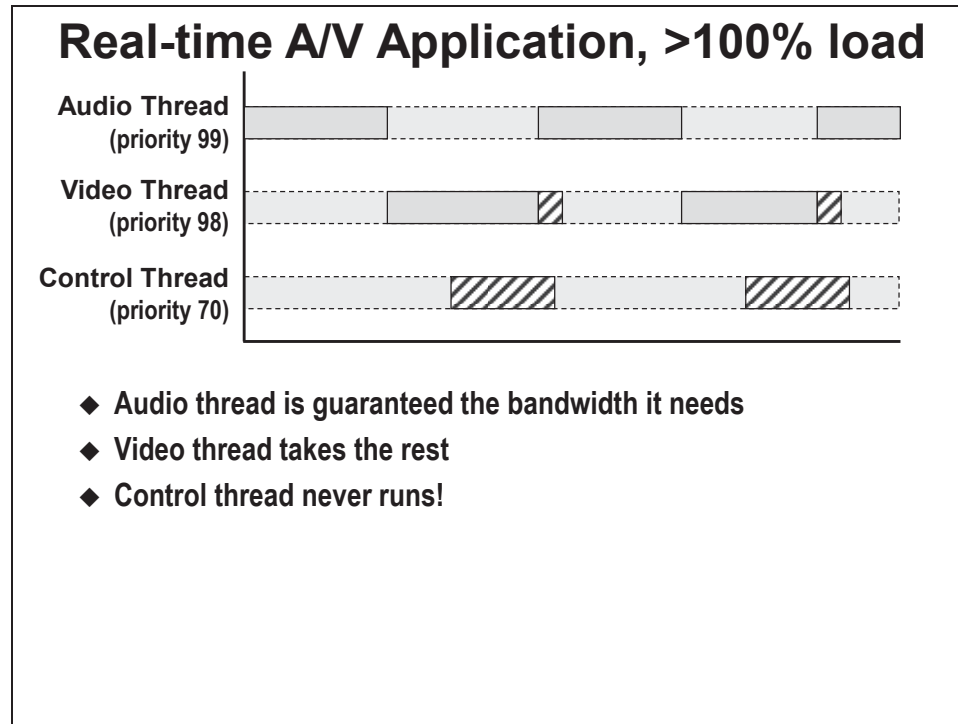
Having each thread distribute the CPU cycles evenly in an overloaded situation may sound like the optimal solution, but let us take a closer look at the effect that each thread missing real time will have on the end-user experience.

First we have the audio thread. The audio thread drops 20% of its samples. This will have a very noticeable effect to the end user. The audio will be highly distorted, perhaps to the point of being unrecognizable, and possibly even painful to the end user. Thus, we will categorize the effect of the dropped audio samples as highly perceptible.

Next we can evaluate the video. The video in this example runs at 30 frames per second, so 6 of these frames are dropped every second. If the video has little motion this may not be noticed. If it has high motion, the end user will probably notice skipping, although it should still be quite possible to see what is happening on the screen. So we will categorize the effect of the dropped video frames as slightly perceptible.

Finally there is the control thread. Because of the control thread only receiving 80% of the CPU it was expecting, the user response is delayed by 1 millisecond. This is not at all perceptible.





So given that mathematically equal treatment has a highly perceptible effect on audio, an intermediate effect on video and almost no effect on the control thread, the system developer might choose to use real-time threads and to rank the priority of these threads according to how perceptible to the end user dropped cycles are for each thread.

Thus, the developer uses the highest priority (99) real-time thread for audio, the next highest priority real-time thread for video, and a relatively low priority real-time thread for control.

In this case, the developer has solved one problem but introduced a new problem. As is seen on the diagram, because the audio thread and video thread together still use more than 100% of the CPU, the control thread never runs for even a single cycle, a condition referred to as lockout or thread starvation.

## Time-Sliced A/V Application Analysis

Audio Thread 

Audio thread completes, no distortion

Video Thread 

Video thread drops 1 of 30 frames

Control Thread 

No user response

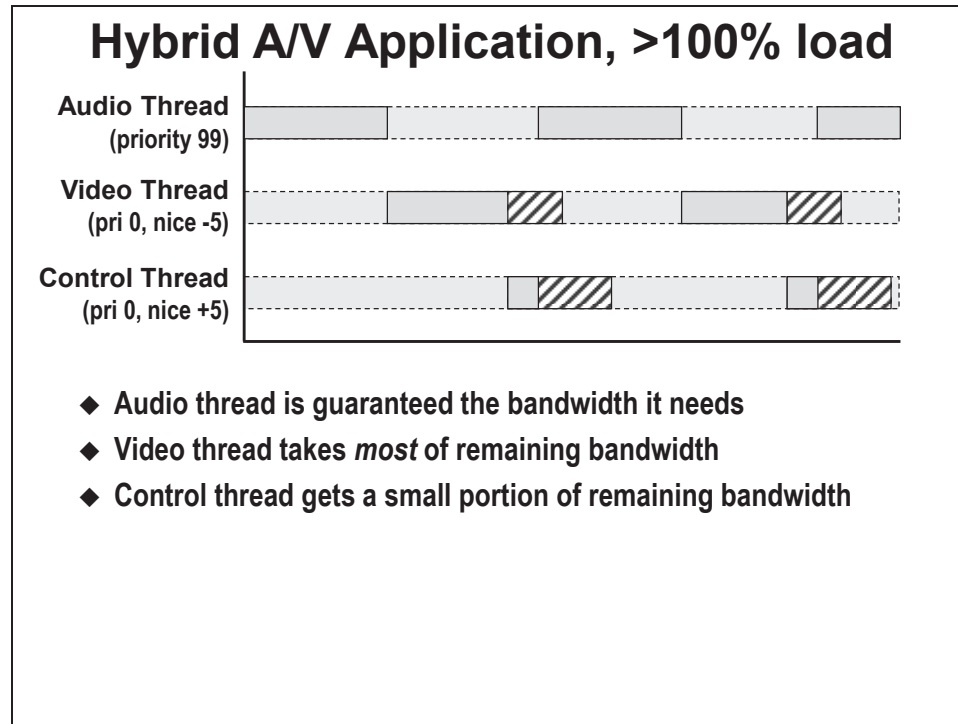
### Still a problem:

- ◆ Audio thread completes as desired
- ◆ Video thread failure is practically imperceptible
- ◆ Control thread never runs – User input is locked out

In our analysis, the audio thread, because it is the highest priority real-time thread in the system is guaranteed to receive all the cycles it requires (assuming that this is less than 100% of the CPU.) So this thread completes with no distortion.

The video thread is also improved. Instead of dropping six video frames, it is now dropping only 1 video frame.

However, the control thread, being the lowest priority thread in the system, receives no CPU cycles. As such, user input is not just delayed, but is completely lost. So this is not a good solution for the system either.



Fortunately, Linux allows intermixing of time-slice and real-time threads. When the system contains both time-slice and real-time threads, all real-time threads in the system run until they have blocked themselves, and then whatever amount of CPU remains is time-sliced between the remaining time-slice threads.

Keep in mind that this means that real-time threads may still lock out time-slice threads if the real-time threads consume 100% of the CPU cycles. Just making a thread time-slice does not guarantee that it will be given CPU cycles. But so long as the system designer can guarantee that all real-time threads together will not consume 100% of the CPU cycles, there will always be something remaining for the time-slice threads.

For this system, the audio thread is made into a real-time thread. The video and control threads are both made into time-slice threads. Because the audio thread will never consume 100% of the CPU cycles, there will always be some amount of CPU cycles remaining for the video and control threads to split. The system designer may even take advantage of the niceness values to skew more CPU cycles towards the video thread from the control thread.

## Hybrid A/V Application Analysis

Audio Thread 

Audio thread completes, no distortion

Video Thread 

Video thread drops 2 of 30 frames

Control Thread 

User response delayed 100ms

### A good compromise:

- ◆ Audio thread completes as desired
- ◆ Video thread failure is barely perceptible
- ◆ Control thread delayed response is acceptable
- ◆ Bottom Line: We have designed the system so that it degrades gracefully

The result is a system that distributes CPU cycles in an effective manner and degrades gracefully even as the system becomes extremely overloaded.

## Default Thread Scheduling

```
#include <pthread.h>
...
pthread_create(&myThread, NULL, my_fxn,
               (void *) &audio_env);
```

- ◆ Setting the second argument to **NULL** means the pthread is created with default attributes

| pThread attributes:       | NULL / default value:      |
|---------------------------|----------------------------|
| stacksize                 | PTHREAD_STACK_MIN          |
| ...                       | ...                        |
| detachedstate             | PTHREAD_CREATE_JOINABLE    |
| schedpolicy               | SCHED_OTHER (time slicing) |
| inheritsched              | PTHREAD_INHERIT_SCHED      |
| schedparam.sched_priority | 0                          |

If a NULL pointer is passed as the pointer to the attribute structure for `pthread_create`, the thread will be created with default values. The default value for the scheduling policy is “SCHED\_OTHER” which is time slicing. So if the programmer wishes to use real-time threads, it will require specifying a real-time thread type in the attribute structure.

## Scheduling Policy Options

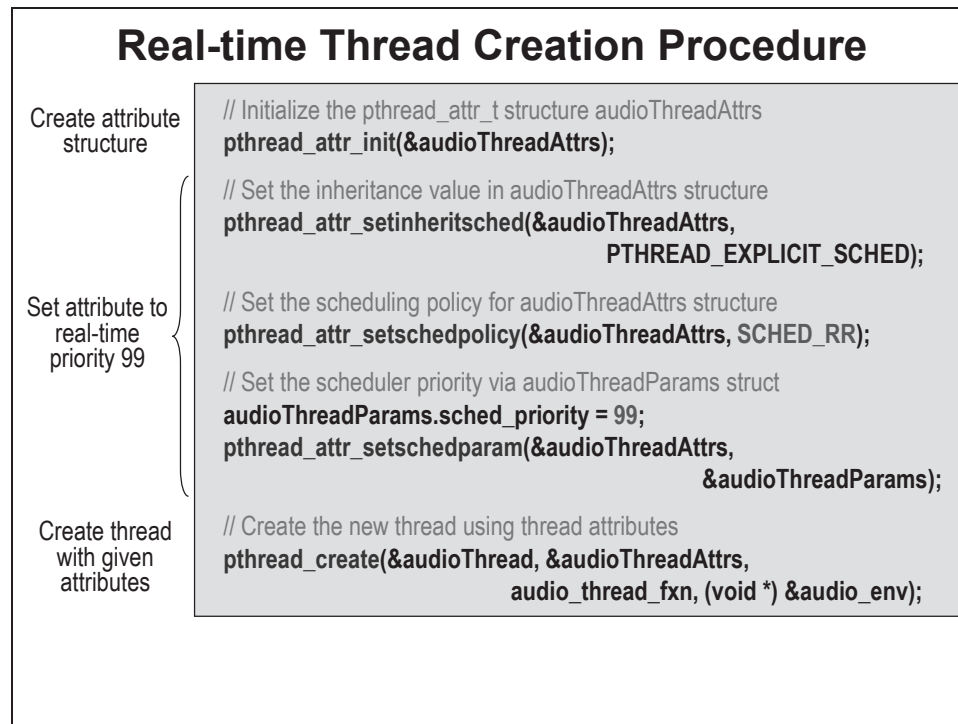
|              | SCHED_OTHER  | SCHED_RR       | SCHED_FIFO |
|--------------|--------------|----------------|------------|
| Sched Method | Time Slicing | Real-Time (RT) |            |
| RT priority  | 0            | 1 to 99        | 1 to 99    |
| Min niceness | +20          | n/a            | n/a        |
| Max niceness | -20          | n/a            | n/a        |
| Scope        | root or user | root           | root       |

- ◆ Time Sliced scheduling is specified with **SCHED\_OTHER**:
  - Niceness determines how much time slice a thread receives, where higher niceness value means less time slice
  - Threads that block frequently are rewarded by Linux with lower niceness
- ◆ Real-time threads use preemptive (i.e. priority-based) scheduling
  - Higher priority threads always preempt lower priority threads
  - RT threads scheduled at the same priority are defined by their policy:
    - ◆ **SCHED\_FIFO**: When it begins running, it will continue until it blocks
    - ◆ **SCHED\_RR**: "Round-Robin" will share with other threads at it's priority based on a deterministic time quantum

There are two types of real-time thread supported by the Linux scheduler. Their behavior is very similar. The only difference in behavior occurs when there are two threads of exactly the same priority that are ready to run.

For the “SCHED\_RR” (round robin) type, the two threads that are at exactly the same priority will share CPU by executing in a round-robin manner. Each thread will run for a deterministic period of time (based on the “time quantum” of the scheduler) in a circular queue.

For the “SCHED\_FIFO” (First In First Out) type, whichever thread became ready to run first will run through until it blocks itself.



This slide shows the procedure for creating and initializing a pthread attribute structure and using this attribute structure to create a new POSIX thread.

One important note is the use of “pthread\_attr\_setinheritsched” to set the inheritance for the thread. The default value of thread inheritance is “PTHREAD\_INHERIT\_SCHED,” which indicates that the child thread that is being created will inherit all attributes from the parent thread that created it. Thus, if this field is not changed to “PTHREAD\_EXPLICIT,” any other fields specified in the attribute structure will be ignored because all settings will be inherited from the parent.

## Linking the pthread Library

- ◆ pthread stands for POSIX thread
- ◆ POSIX threads were introduced to Linux in 1996, but are not widely used because of the (relatively) low overhead of using processes.
- ◆ What does POSIX mean?
  - IEEE POSIX committee defined POSIX 1003.1c standard for threads.
  - Linux threads are modeled after this standard, though don't completely adhere to it.

```
#include <pthread.h>

#include          C compiler pre-processor statement
pthread.h        Function declarations for the posix thread library

# gcc -D_REENTRANT myProg.c -o myprog.o -lpthread
gcc              GNU compiler collection. Invokes compiler,
                  assembler and linker as necessary for specified
                  source files.

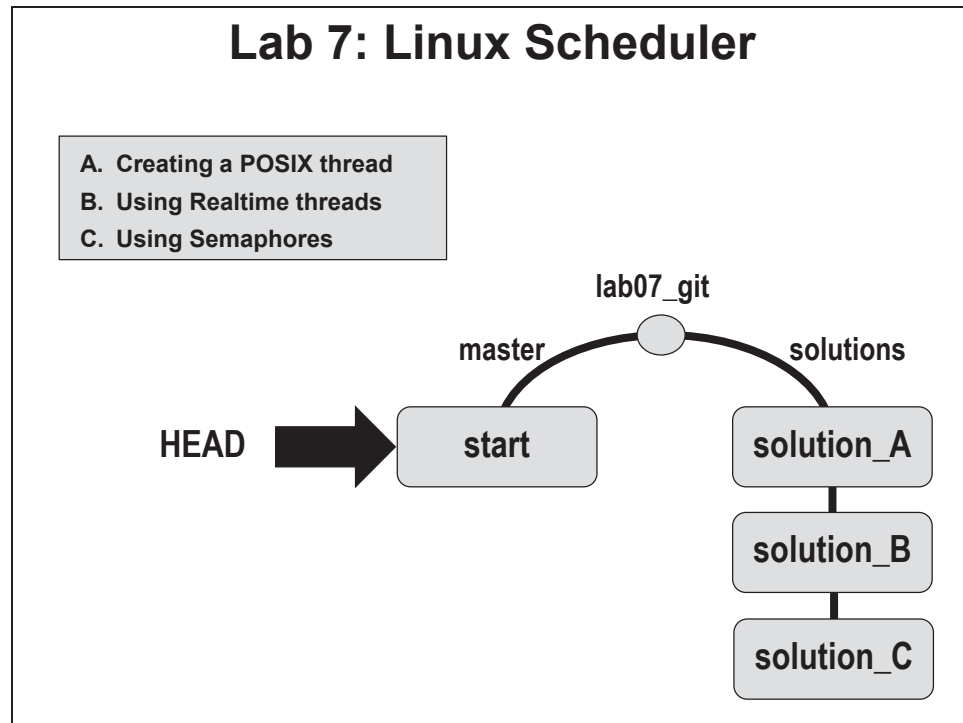
-D_REENTRANT     Defines the _REENTRANT symbol, which forces
                  all included libraries to use reentrant functions.

-lpthread        Link in the pthread (POSIX thread) library.
```

All of the pthread functions utilized in this module are contained in the pthread library, libpthread.so (shared) or libpthread.a (static.) In addition to linking the library's header file, pthread.h, application authors need to link the pthread library with the “-lpthread” flag in gcc.

Another optional flag that many users will find useful is the “-D\_REENTRANT” flag. IF this flag is set, then only reentrant functions will be linked from the standard libraries. While not an absolute requirement, using only reentrant functions greatly simplifies the debugging of multi-threaded applications.

## Lab 7: Linux Scheduler



### Lab 7A: Timeslice Thread

- A. Creating a POSIX thread  
 B. Using Realtime threads  
 C. Using Semaphores

**Linux scheduler shares CPU between timesliced threads, so each thread gets a turn to run.**

```

Console
<terminated> 06_pthread_lab f
Remote debugging from host 192.168.1.1
Creating thread 1
Creating thread 2
Entering thread #1
Inside while loop of thread #1

All application threads started
Entering thread #2
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Inside while loop of thread #2
Inside while loop of thread #1
Exiting thread #1
Exiting thread #2
Exited thread #1
Exited thread #2

Child exited with status 0
GDBserver exiting
  
```



## Lab 7B: Realtime Thread

- A. Creating a POSIX thread
- B. Using Realtime threads**
- C. Using Semaphores

The first realtime thread to run will lock out the second thread.

Because delay function uses a for loop instead of sleep (blocking)

```

Console
<terminated> 06_pthread_lab
Remote debugging from host 192.168.1.1
Creating thread 1
Entering thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Creating thread 2

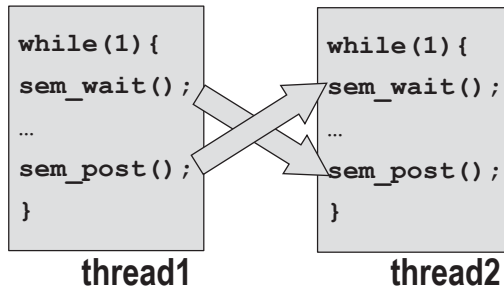
All application threads started
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Exiting thread #1
Entering thread #2
Exiting thread #2
Exited thread #1
Exited thread #2

Child exited with status 0
GDBserver exiting

```

## Lab 7C: Realtime + Semaphore

- A. Creating a POSIX thread
- B. Using Realtime threads
- C. Using Semaphores**



```

Console
<terminated> 06_pthread_lab
Remote debugging from host 192.168.1.1
Creating thread 1
Entering thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Creating thread 2

All application threads started
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Inside while loop of thread #1
Exiting thread #1
Entering thread #2
Exiting thread #2
Exited thread #1
Exited thread #2

Child exited with status 0
GDBserver exiting

```

(This page intentionally left blank)

# Module 08: Networking

---

## Introduction

Networking connectivity is a common requirement for many applications. Linux on the AM335x provides strong support for both wired and wireless IP communication. This module begins with a short review of basic networking concepts and then proceeds into a discussion of various commonly used server/client protocols such as secure shell (ssh), telnet, file transfer protocol (ftp), dynamic host configuration protocol (dhcp) and dynamic domain name servers (ddns.)

The module ends with an in-depth study of the underlying protocol common to all of these server/client systems in a Linux environment: Berkeley sockets. Programming examples are shown for opening and closing Berkeley sockets on both the server and client.

## Module Topics

|                                      |             |
|--------------------------------------|-------------|
| <b>Module 08: Networking .....</b>   | <b>8-1</b>  |
| <i>Module Topics.....</i>            | <i>8-2</i>  |
| <i>Ethernet Basics .....</i>         | <i>8-3</i>  |
| <i>Server/Client Protocols.....</i>  | <i>8-8</i>  |
| <i>Berkeley Sockets.....</i>         | <i>8-11</i> |
| <i>Lab 8: Berkeley Sockets .....</i> | <i>8-15</i> |

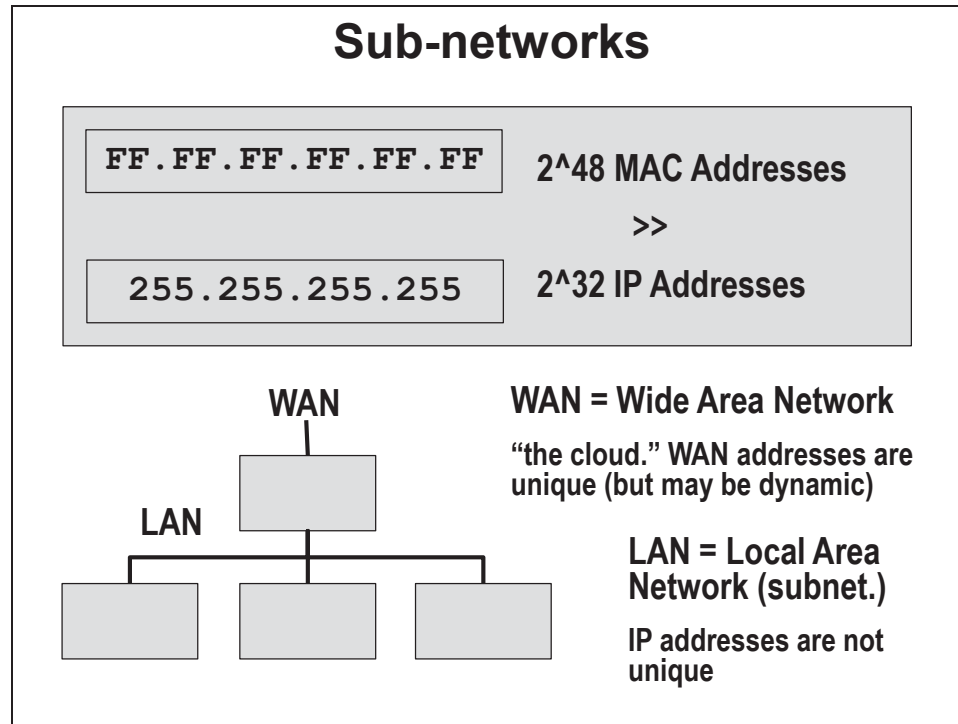
# Ethernet Basics

| Networking References   |   |
|---|---|
| <b>FF.FF.FF.FF.FF.FF</b>  | <b>Media Access Controller address is unique to a physical device</b> |
| <b>255.255.255.255</b>  | <b>IP address is assigned by network or network administrator</b>     |
| <b>www.ti.com</b>   | <b>Host name and domain name are resolved by domain name servers</b>  |
| <b>Address Resolution Protocol (ARP) resolves MAC addresses from IP addresses</b> |   |
| <b>Domain Name Service (DNS) resolves IP addresses from host and domain name</b>  |   |

There are three ways for a machine to be referenced on the internet. The most fundamental address is the Media Access Controller (MAC) address. MAC addresses are unique and must be purchased from IEEE.

If the device is running the Internet Protocol (IP) then it has an IP address that is either statically assigned by a network administrator or dynamically assigned by the network using the Dynamic Host Configuration Protocol (DHCP.) IP addresses are unique only on the Wide Area Network (WAN), also known as “the cloud.” Sitting off of the WAN backbone are numerous subnetworks.

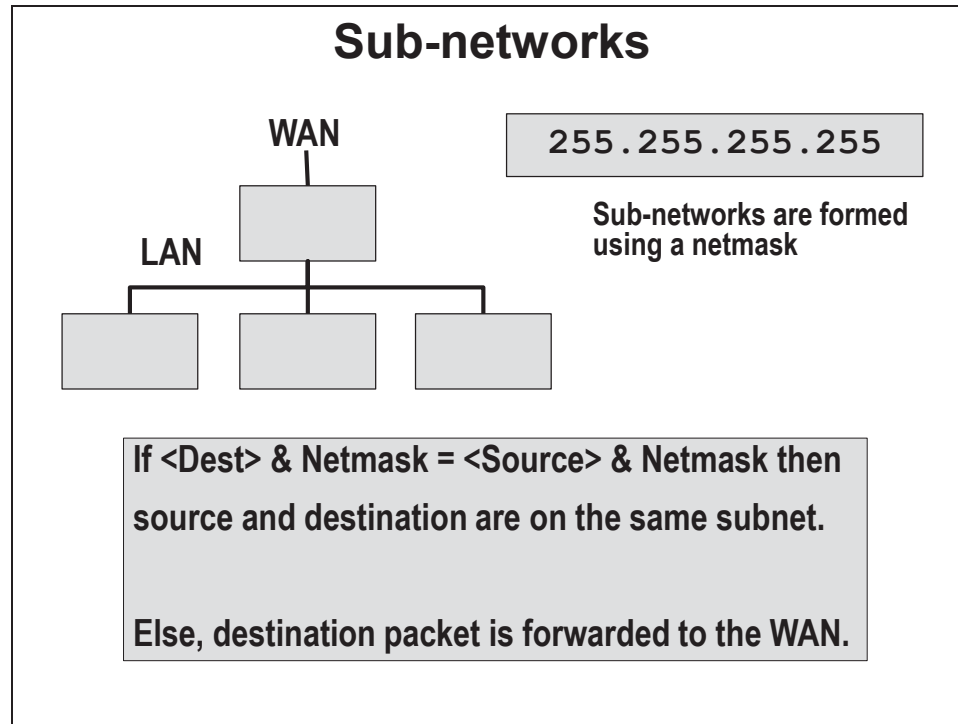
Finally, machines may be assigned a host name, which is translated to an IP address by Domain Name Servers, which are basically databases of host name to IP address translations.



There are 2 to the power of 48 possible MAC addresses, but only 2 to the power of 32 possible IP (version 4) addresses. This is possible because, though MAC addresses are unique across the world, IP addresses are only unique on the Wide Area Network, i.e. “the cloud.” Numerous subnetworks, also known as “Local Area Networks” (LANs) are attached to the Wide Area Network. Within a LAN subnetwork, IP addresses must be unique, but they are not unique across LAN subnetworks.

For instance, many home networks use a router to connect to the Wide Area Network via cable modem, DSL or other connection type. Most routers default to an IP address of 192.168.1.1, 192.168.0.1 or 10.0.0.1

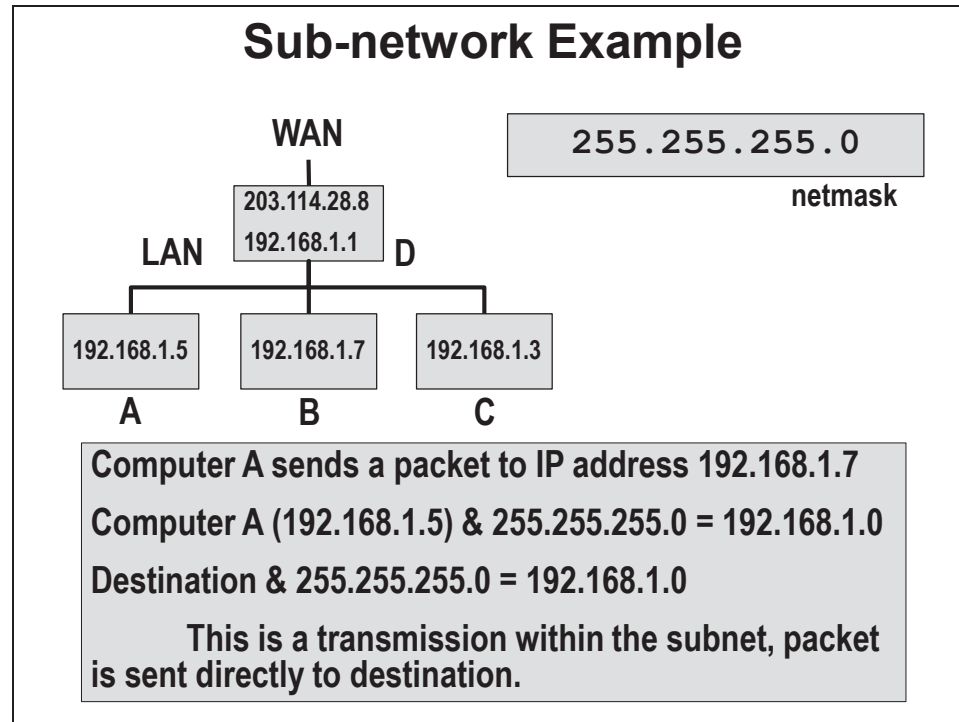
Obviously there are millions and millions of routers in the world, so three IP addresses are hardly sufficient for all routers to have a unique IP address. The IP address mentioned is, however, the router’s LAN IP address. No other device shares this IP address on the LAN, so the IP address is unique on the LAN, i.e. across the user’s home network. The IP address just isn’t unique from one home network to the next.



A subnetwork, i.e. a Local Area Network, is formed using a netmask. A netmask is specified as four octets (8-digit binary numbers) which each range from 0-255. By taking a computer's IP address and using a bitwise-and operation with this IP address and the computer's netmask, one determines the subnetwork address.

The destination IP address that a computer sends an IP packet to is on the same subnetwork as the sending computer if the bitwise and operation of the netmask and destination IP address generates the same subnetwork address.

If the sending computer and destination IP are on the same subnetwork, the packet will be sent directly. If, however, the destination IP is on a different subnetwork, then the packet will be forwarded to the Wide Area Network via a gateway.



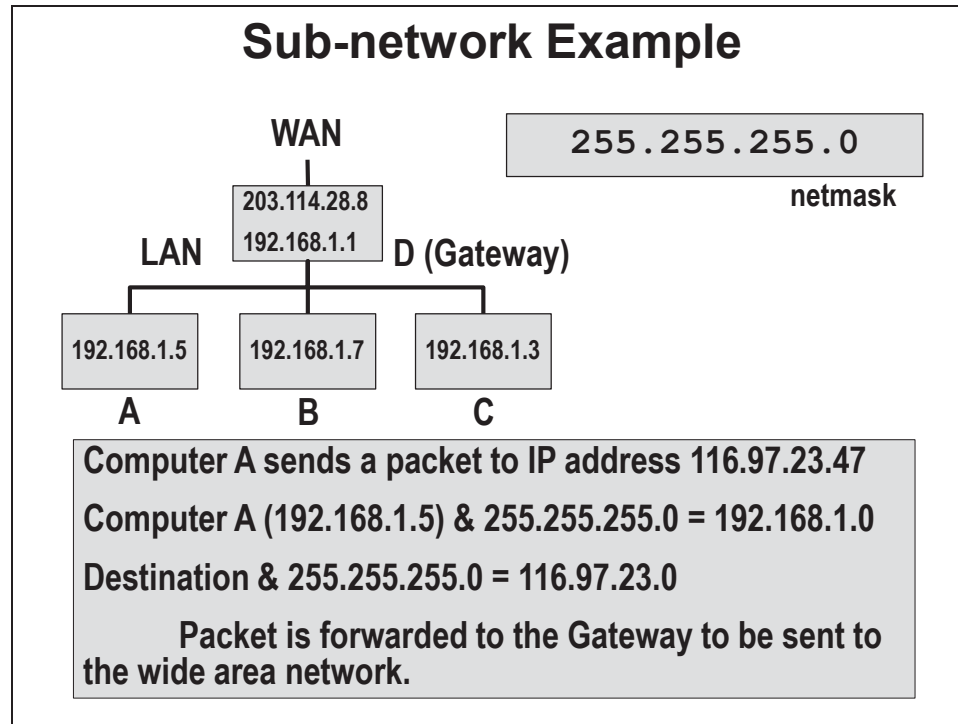
In this example there is a gateway, such as a router, at position D. The gateway has both a LAN IP address that it uses to connect to the subnetwork and a WAN IP address that it uses to forward packets from the LAN to the WAN.

The LAN IP address of the gateway is 192.168.1.1, with a netmask of 255.255.255.0

Using a bitwise and operation between these two quantities, the subnetwork address defined by the gateway is 192.168.1.0, and any computer with an address 192.168.1.XXX is on the subnetwork where XXX includes 1...255.

Thus if Computer A at IP address 192.168.1.7 sends a packet to 192.168.1.5, then, because both IP addresses when bitwise and-ed with the netmask produce the same result (192.168.1.0), these computers are on the same subnetwork and the packet is sent directly to the destination.





In the second example on the same subnetwork, computer A at IP address 192.168.1.5 sends a packet to 116.97.23.47

When bitwise and-ing computer A's IP address, the result is 192.168.1.0, which as previously stated is the subnetwork address.

When bitwise and-ing the destination IP address, the result is 116.97.23.0

Since this is different than the address of computer A's subnetwork, computer A knows that this destination address is not on the local subnetwork and must forward the packet to the gateway address, which is 192.168.1.1

The gateway, which receives the packet on the LAN IP connection at 192.168.1.1 will then forward the packet out to the Wide Area Network via its WAN IP connection at 203.144.28.8

## Server/Client Protocols

### IP Ports

In addition to an IP address, connection requests may specify a port number.

Ports are generally used in order to route connection requests to the appropriate server.

| Service    | Port  |
|------------|-------|
| echo       | 7     |
| ftp        | 20/21 |
| ssh        | 22    |
| telnet     | 23    |
| nameserver | 42    |
| http       | 80    |

#### Client/Server model

**Server is a static background (daemon) process, usually initiated at startup, that listens for requests from one or more clients.**

A fundamental concept in the IP network model is the concept of client to server connections. In the client/server model, servers exist on the network and are constantly listening for a remote client to initiate a connection. Theoretically these servers are listening on the network 24 hours a day, 365 days a year. One example of a well-known server would be the http server located at [www.ti.com](http://www.ti.com), which is constantly alive on the network, waiting for a web browser to request a connection. In this example, the web browser is the client.

A client, by comparison, does not require constant access to the IP network. A client needs only to have IP connectivity during the span of its live connection to a given server. An example would be a web browser, which need only have network access while browsing web pages.

In many configurations, a single IP address (often a single physical device) is used to provide multiple servers, such as a nameserver, and echo server, a secure shell server, etc. How then does the system route client requests to the appropriate service. This is accomplished via port numbers. In addition to specifying an IP address, which is generally sufficient to route a packet to a given physical device, the packet address contains a port number. Each server is assigned a unique port number so that a client may attach to a specific service (using the port number) on a specific device (using IP address.) This requires agreement between client and server on the port number that will be used for communication. Generally, services on the cloud use a standard set of port numbers as shown on the diagram. For instance, http servers are often configured to respond to port 80.

## Dynamic Host Configuration Protocol (DHCP)

DHCP server issues IP address, netmask, gateway, dns to DHCP clients.

Server installation (Ubuntu)  
**# apt-get install dhcp3-server**  
 Server configuration file  
**/etc/dhcp3/dhcpd.conf**

```

user@ubuntu: ~
File Edit View Terminal Help
user@ubuntu:~$ sudo dhclient eth0
Internet Systems Consortium DHCP Client V3.1.3
Copyright 2004-2009 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/

Listening on LPF/eth0/00:0c:29:ee:d6:7e
Sending on LPF/eth0/00:0c:29:ee:d6:7e
Sending on Socket/fallback
DHCPDISCOVER on eth0 to 255.255.255.255 port 67 interval 3
DHCPOFFER of 192.168.1.112 from 192.168.1.1
DHCPREQUEST of 192.168.1.112 on eth0 to 255.255.255.255 port 67
DHCPACK of 192.168.1.112 from 192.168.1.1
bound to 192.168.1.112 -- renewal in 38152 seconds.
user@ubuntu:~$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0c:29:ee:d6:7e
          inet addr:192.168.1.112  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:feee:d67e/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:38 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:7718 (7.7 KB)
          Interrupt:19 Base address:0x2024
  
```

One of the most commonly used services on an IP network is the Dynamic Host Configuration Protocol, or DHCP. This protocol is used to assign IP addresses dynamically to computers as they attach to the network. Because so many devices connect wirelessly to various “hotspots,” a static IP configuration is difficult, and dynamic IP addresses have become so common that even networks that are effectively static in terms of their physical connections often use DHCP because it reduces the effort required by a network administrator to assign and maintain static addresses.

When a DHCP-enabled computer attaches to an IP network, its DHCP client sends a broadcast message. The DHCP server then responds and after a handshake, assigns a dynamic IP address to the client. An example of a common DHCP server would be most home routers, which are often configured to assign IP addresses dynamically.

## Domain Name Service

Domains and subdomains are not required to correlate to subnetwork boundaries (although they often do.)

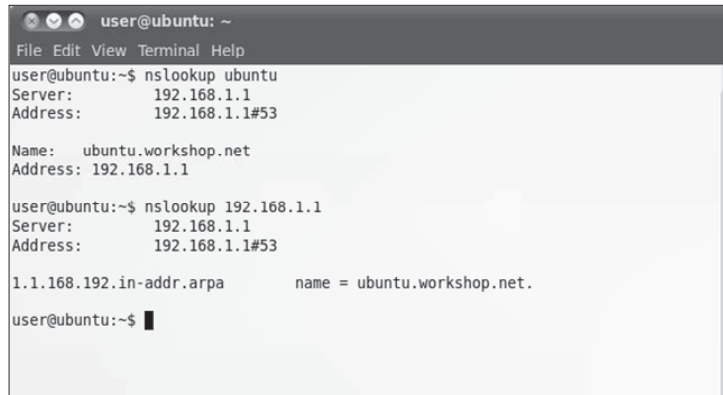
### Server installation (Ubuntu)

```
# apt-get install bind9
```

Configuration files

```
/etc/named.conf
```

```
/var/bind/*.zone
```

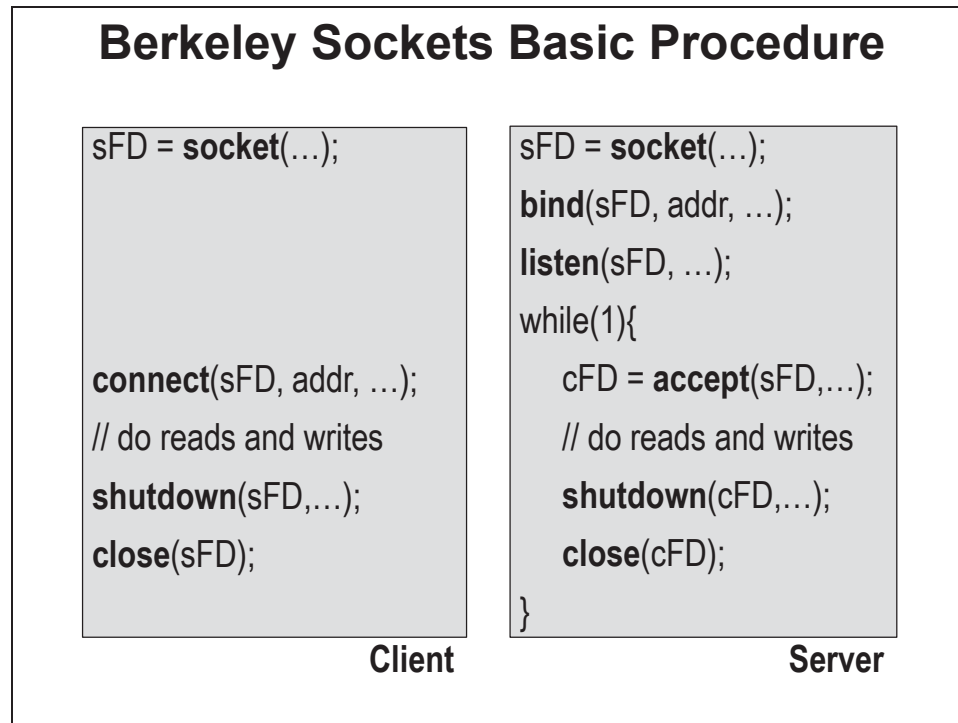


```
user@ubuntu: ~  
File Edit View Terminal Help  
user@ubuntu:~$ nslookup ubuntu  
Server:      192.168.1.1  
Address:     192.168.1.1#53  
  
Name:   ubuntu.workshop.net  
Address: 192.168.1.1  
  
user@ubuntu:~$ nslookup 192.168.1.1  
Server:      192.168.1.1  
Address:     192.168.1.1#53  
  
1.1.168.192.in-addr.arpa      name = ubuntu.workshop.net.  
  
user@ubuntu:~$
```

Another common client/server utility on IP networks is the domain name service, or DNS. A DNS server is a database that contains entries to translate human-readable “host names” into IP addresses, which is termed forward lookup, and to translate IP addresses into host names, which is termed reverse lookup.

When a user enters “www.ti.com” into a http browser, this human-readable text is converted into the appropriate IP address from a Domain Name Server on the network.

# Berkeley Sockets



One of the reasons that nearly every protocol that exists on IP networks is based on a client/server model is that most of these protocols are written on top of a Berkeley Sockets transmission layer, and Berkeley Sockets is based on a client/server model.

This slide shows the basic outline of a typical Berkeley Sockets client application as well as server application. The gaps in the client outline are there to align client calls to similar or corresponding function calls in the server.

The procedure for initiating a connection with a Berkeley Sockets client is fairly straightforward. The first step is to declare a socket using the “socket” call. This function returns a file descriptor. The connect call, which takes the socket file descriptor as its first parameter, then connects the socket to a remote IP address and port using the “connect” function. If the connection is successful, the user may then read and write across the connection via the socket file descriptor, using exactly the same function call that would be used to read and write across a UART connection using its file descriptor. Once the program has finished communicating, it can send a “shutdown” message over the network. This will alert the server that the client is attached to that it is shutting down the connection. Finally, the close command releases the connection.

The procedure for creating a server application is only slightly more complex. Just as with the client, the first step is to declare a socket with the socket call. The server then uses a combination of “bind” and “listen” calls to first bind the socket to a local IP address and port, and then listen for client connection requests on that IP address and port. Note that at this point in the application no connection has been accepted by the server. The “listen” call configures a listen queue, and as connection requests are received from clients, they are placed onto the listen queue. The server can then accept a connection from the listen queue using the “accept” call, which returns a new file descriptor, the connection file descriptor. This file descriptor may be used to read and write, and eventually to shutdown and close the connection, just as with the client.

## Socket Types

### Local Socket

```
sFD = socket(AF_UNIX, SOCK_STREAM, 0);
```

### TCP/IP Socket

```
sFD = socket(AF_INET, SOCK_STREAM, 0);
```

### UDP/IP Socket

```
sFD = socket(AF_INET, SOCK_DGRAM, 0);
```

**Final parameter to socket function is protocol.  
Protocol is typically determined completely by first  
two parameters, so that “0” (default) is usually passed.**

The declaration of the socket determines the socket type and therefore the transmission protocol used. Not all Linux sockets are actually IP sockets. For instance, the “AF\_UNIX” socket declaration is used to create a local socket that can be used to communicate between processes on a single Linux device without the use of an IP stack.

For IP communication, two socket types are supported. The “SOCK\_STREAM” declaration will configure the socket to use the TCP protocol, whereas “SOCK\_DGRAM” will configure the socket to use UDP.

## Specifying Address and Port

```
struct sockaddr_in stSockAddr;  
  
memset(&stSockAddr, 0, sizeof(stSockAddr));  
  
stSockAddr.sin_family = AF_INET;  
stSockAddr.sin_addr.s_addr = inet_addr("192.168.1.1");  
stSockAddr.sin_port = htons(1100);
```

### Client

```
connect(SocketFD, (struct sockaddr *)&stSockAddr,  
        sizeof(stSockAddr));
```

### Server

```
bind(SocketFD, (struct sockaddr *)&stSockAddr,  
      sizeof(stSockAddr))
```

The IP address and port are specified in a single structure of type “sockaddr\_in.” Note the usage of the “inet\_addr” macro that will convert an IP address written as a human-readable string into the appropriate binary representation. Also, the “htons” (host to network short) macro should be used when specifying the port number. This macro will ensure that the endianness of the port number is correct before it is placed into the IP packet.

The client will fill in this structure and use it to specify the IP address and port of the server that it is connecting to. The server will use this structure to specify the IP address and port that it is listening on.

## Server Details

```
listen(int sFD, int backlog);
```

**listen creates an incoming connection queue**

**socketFD is the file descriptor**

**backlog is the maximum number of connection requests to hold in the queue**

```
accept(int sFD, sockaddr *address, size_t *len);
```

**address and len (of address struct) are used to return the address of the connecting client. If this is not of interest, NULL may be passed.**

```
shutdown(int sFD, int how);
```

**how may be SHUT\_RD, SHUT\_WR, SHUT\_RDWR**

**possible to call with SHUT\_RD followed by SHUT\_WR**

**shutdown informs TCP/IP stack to terminate session, but does not close file descriptor. Many systems implement shutdown automatically in close, but best to always call.**

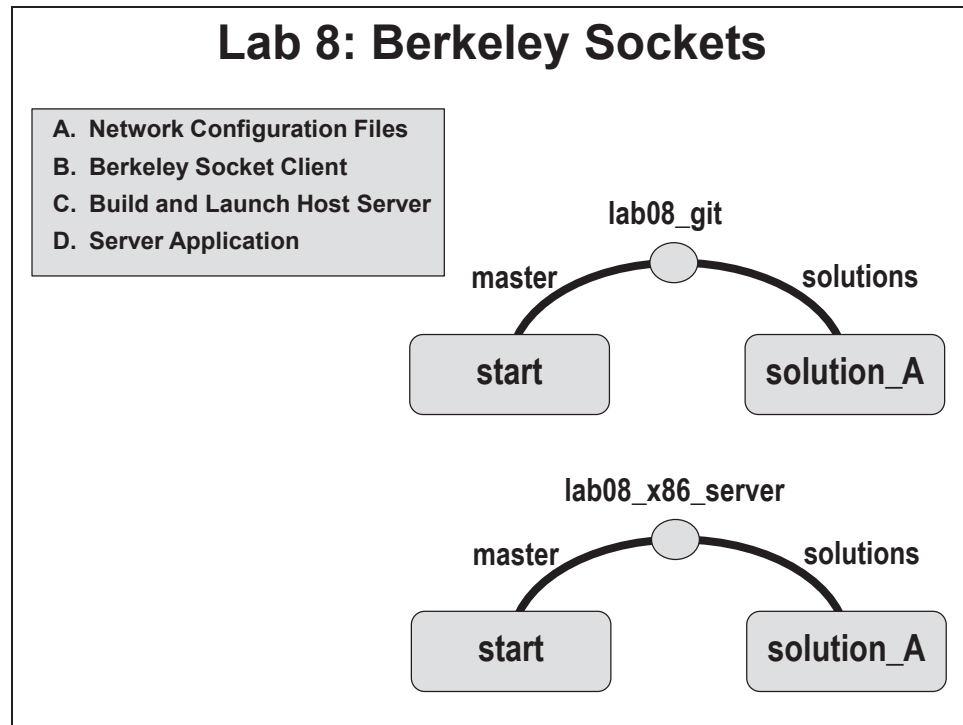
Here are the parameter details of the final three functions. The listen call specifies an integer backlog. This is the number of backlogged requests that the listen queue will hold before the server refuses incoming connection requests. A typical number is 10.

The accept call allows the user to specify an address structure, passed by pointer, and the size of that address structure, the latter value used to ensure that the function does not overwrite the structure. If a NULL pointer is passed, then the field is ignored. If, however, a valid pointer to an address structure is passed, the accept function will fill the structure in with the IP address of the client that is initiating the connection request. This IP address is not required by the application to respond to the client, the application uses the file descriptor returned by the accept call for this purpose. The information is provided in case the server wishes to log the IP address of clients that are initiating connection requests.

The shutdown command takes a “how” variable that allows the user to shutdown either just the read channel, just the write channel, or to shut down both the read and write channels.



## Lab 8: Berkeley Sockets



(Page intentionally left blank)

# Module 09: Video and Graphics Drivers

---

## Introduction

Linux developers commonly use two drivers for handling video and graphics. The first of these drivers is named “Video For Linux, 2” (V4L2) and is commonly used to capture or display moving video. The second is called the “Framebuffer Device” driver (FBDEV) and is commonly used to display graphics such as a graphical user interface or desktop environment, although it is possible to use FBDEV to display moving video as well.

This module will cover functionality overviews as well as application programming details of using the V4L2 and FBDEV drivers.

## Module Topics

|   |             |
|---|-------------|
| <b>Module 09: Video and Graphics Drivers.....</b> | <b>9-1</b>  |
| <i>Module Topics.....</i>                         | <i>9-2</i>  |
| <i>Linux v4L2 Video Driver.....</i>               | <i>9-3</i>  |
| <i>Using mmap.....</i>                            | <i>9-8</i>  |
| <i>Linux FBdev Display Driver.....</i>            | <i>9-13</i> |
| <i>Lab Exercise .....</i>                         | <i>9-21</i> |

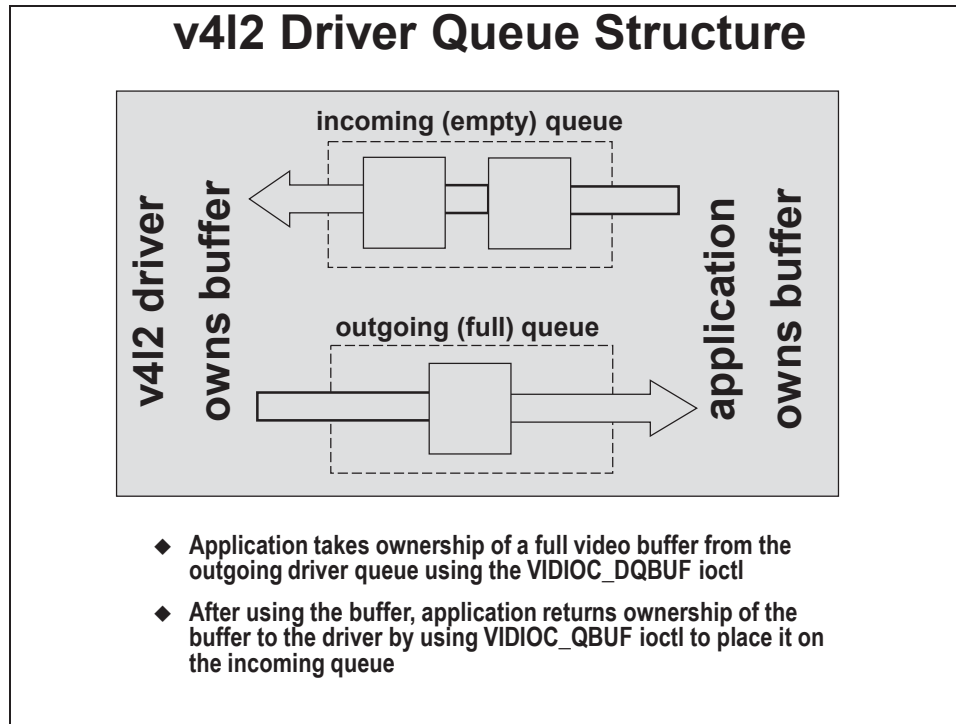
# Linux v4L2 Video Driver

## v4l2 Driver Overview

- ◆ **v4l2 is a standard Linux video driver used in many linux systems**
- ◆ **Supports video input and output**
  - In this workshop we use it for video input only
- ◆ **Device node**
  - Node name: `/dev/video0`
  - Uses major number 81
- ◆ **v4l2 spec:** <http://bytesex.org/v4l>
- ◆ **Driver source location:**  
`.../lsp/ti-davinci/drivers/media/davinci_vpfe.c`

The “Video For Linux, iteration 2” driver is a standard Linux driver that TI has ported to the Sitara platform for use with the Arago distribution. The standard device node name is “/dev/videoX” where X is an integer enumeration of video devices, beginning with 0. The driver generally uses a statically declared major number of 81.

v4l2 is optimized for high frame-rate (24, 30, 50, 60 fps, etc.) video, and can be used for either video capture or display.



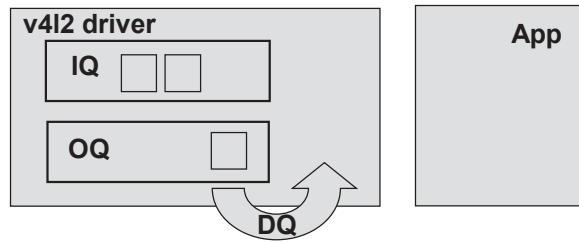
Because v4l2 is made for high frame rate video, it requires a multi-buffer scheme. “Ping-Pong” buffers are commonly used in real-time systems to allow the application to update one buffer (i.e. “Ping”) while a second buffer is being displayed (i.e. “Pong”)

Some applications will use more than two buffers. In some cases an extra buffer is used to absorb real-time jitter. In other cases the application requires access to multiple buffers simultaneously, as with the H.264 video compression standard, which uses both forward and backward encoding of frames.

In order to create a driver that could be used with a flexible number of buffers, the v4l2 authors created a double-queue system. The queues can hold an arbitrary number of buffers and the bi-directional configuration of the two queues allows these buffers to be continuously circulated between the application and the driver.

From the application standpoint, the “`VIDIOC_DQBUF`” ioctl command is used to dequeue a buffer from the outgoing queue, and the “`VIDIOC_QBUF`” ioctl command is used to return the buffer to the incoming queue after use. In the diagram above, a capture driver is shown, so the buffers on the outgoing queue contain valid newly-captured data and are labelled “full,” whereas the buffers on the return queue are labelled “empty” because the data they contain is stale and will be overwritten by the capture driver. For a display driver, this would be reversed.

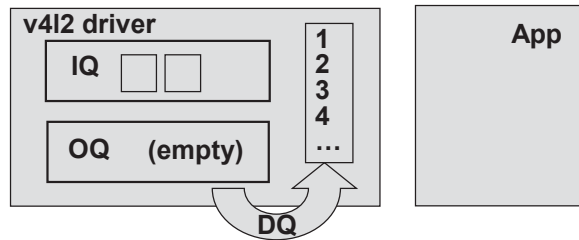
## v4l2 Enqueue and Dequeue



- ◆ Buffers typically exist in driver's memory space
- ◆ The dequeue call makes the data available to the app

Taking a closer look at the mechanics of the v4l2 driver, the first thing to realize is that the video buffers that are queued and dequeued from and to the driver never leave the memory space of the driver. (The memory spaces are designated by the “v4l2 driver” and “App” boxes.) The dequeue operation does not transfer the buffer to the application, it simply transfers ownership of the buffer to the application. Ownership is an important concept with the v4l2 driver, because if the application attempts to read or write a buffer that is owned by the driver, there is a possibility that the driver will be manipulating the buffer simultaneously, which will lead to a corruption of data.

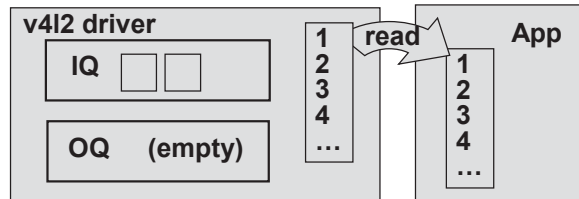
## v4l2 Enqueue and Dequeue



- ◆ Buffers typically exist in driver's memory space
- ◆ The dequeue call makes the buffer available to the app
- ◆ Even after DQ, buffers still exist in the driver's memory space but not the application's

Once the buffer is dequeued, the application owns it, but since the buffer does not exist in the application's memory space, it still cannot access the buffer directly.

## v4l2 Enqueue and Dequeue

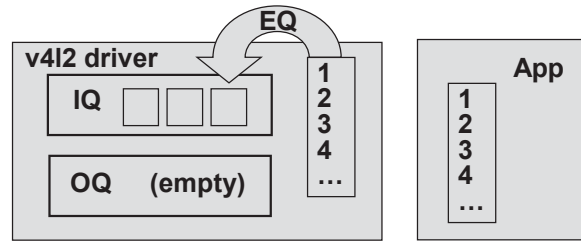


- ◆ Buffers typically exist in driver's memory space
- ◆ The dequeue call makes the buffer available to the app
- ◆ Even after DQ, buffers still exist in the driver's memory space but not the application's
- ◆ read and write operations copy buffers from driver's memory space to app's or vice-versa

The buffer would then use a read call to read the buffer, via the driver's device node, into the application space. Note that this is a copy-based operation. The read request actually transfers control to the Linux kernel, which then copies the driver buffer into the application space. (The kernel has access to the entire memory space of the device.)



## v4l2 Enqueue and Dequeue



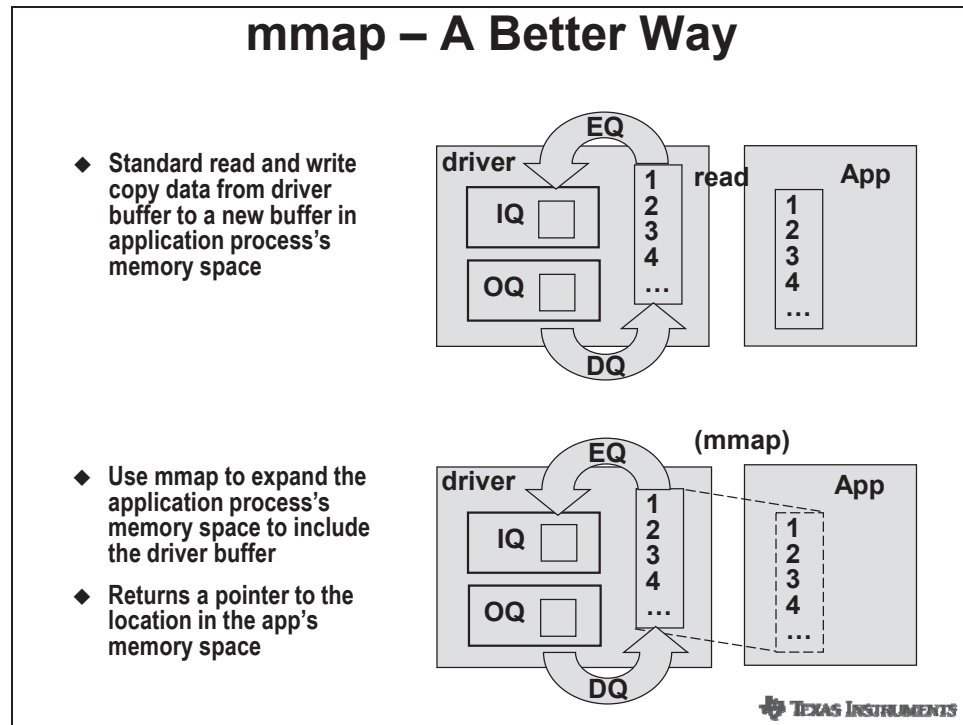
- ◆ Buffers typically exist in driver's memory space
- ◆ These buffers exist in the driver's memory space but not the application's
- ◆ read and write operations copy buffers from driver's memory space to app's or vice-versa

Is there a better method than "read"?

Once a copy of the buffer has been made, the application can enqueue the buffer, passing ownership back to the v4l2 driver. This mechanism correctly handles the arbitration or ownership issue of ensuring that the driver and app do not attempt to access buffers simultaneously, but the copy operation is expensive in terms of both memory and CPU cycles, which for high-resolution, high-frame-rate video could be crippling to system performance.

For this reason, Linux supports a means of directly mapping driver buffers into the memory space of the application.

## Using mmap



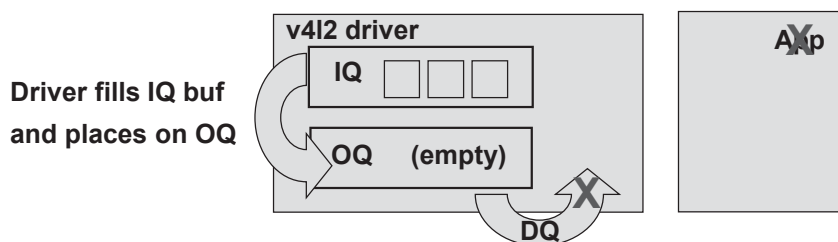
The driver buffers may be mapped into the memory space of the application using a system function call named “mmap” (Memory MAP.)

Before discussing the details, it should be noted that the partitioning of memory spaces using the Memory Management Unit has an important function, which is to protect applications from corrupting each other's memory spaces or corrupting the memory space of the kernel. If a pointer error in an application causes it to overwrite program memory or state variable of the Linux kernel, it will likely crash the entire system.

So bypassing the memory protection feature that is built into the operating system is not an action that should be taken lightly. That said, the mmap function does not actually allow the application to map the buffer into its own space. When the application calls the mmap function, a request is sent to the Linux kernel, which uses the corresponding driver to map the buffer into the application space. This may seem like a trivial distinction, but it is not. The driver is a trusted entity, meaning that it is assumed to be robust due to testing throughout the community, so it should never allow the application to map any memory from kernel space other than the actual data buffer memory. This means that the application might corrupt the data that is stored in these buffers, but the application still does not have access to program memory or internal state variables of the driver or Linux kernel, maintaining this important protection.

Once the data buffers have been mapped into the application space, their contents may be directly read or written, bypassing the overhead of copying driver data back and forth.

## v4l2 Queue Synchronization

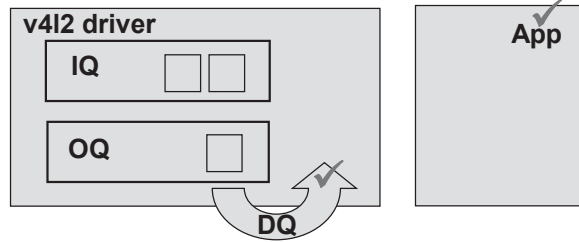


- ◆ The `VIDIOC_DQBUF` ioctl blocks the thread's execution waits if until a buffer is available on the output queue
- ◆ When driver adds a new, full buffer to the output queue, the application process is released
- ◆ Dequeue call completes and application resumes with the following set of commands

In addition to arbitrating ownership between the application and driver, the bi-directional queue structure also provides the important function of synchronization between the application and driver.

If the application attempts to dequeue a buffer from the outgoing queue while the outgoing queue is empty, the ioctl function call will block.

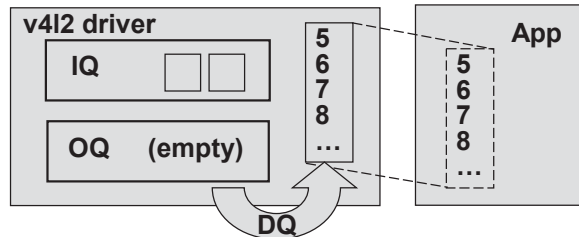
## v4l2 Queue Synchronization



- ◆ The `VIDIOC_DQBUF` ioctl blocks the thread's execution until a buffer is available on the output queue
- ◆ When driver adds a new, full buffer to the output queue, the application process is released
- ◆ Dequeue call completes and application resumes with the following set of commands

The driver continues capturing data, even though the application is blocked, so at some later point a newly filled buffer is placed by the driver onto the outgoing queue.

## v4l2 Synchronization



- ◆ The `VIDIOC_DQBUF` ioctl blocks the thread's execution until a buffer is available on the output queue
- ◆ When driver adds a new, full buffer to the output queue, the application process is released
- ◆ Dequeue call completes and application resumes with the following set of commands

When the driver places a newly filled buffer on the outgoing queue, this releases the block on the application, which allows the application to complete the dequeue operation. If mmap has been used to map the buffer into the application space it may then access the data.

## Commonly Used v4l2 ioctls

### Data Structures

```
struct v4l2_requestbuffers req;
req.count;      // how many buffers to request
req.type;       // capture, output, overlay
req.memory;     // mmap, userptr, overlay

struct v4l2_buffer buf;
buf.index;      // which driver buffer
buf.type;       // matches req.type
buf.memory;     // matches req.memory
buf.m.offset;   // location of buffer in driver mem
```

### Request the driver allocate a new buffer

```
ioctl(fd, VIDIOC_REQBUFS, &req);
```

### Get information on a driver buffer

```
ioctl(fd, VIDIOC_QUERYBUF, &buf);
```

### Enqueue and Dequeue buffers to/from driver

```
ioctl(fd, VIDIOC_QBUF, &buf);
Ioctl(fd, VIDIOC_DQBUF, &buf);
```

Here are some of the commonly used structure elements and function calls defined in the v4l2 library and associated header file. The usual operation is for the driver to allocate its own buffers, as opposed to the application allocating them. This is due in part to the fact that most drivers require physically continuous (not just virtually continuous) buffers because many drivers utilize hardware such as DMA channels that bypass the Memory Management Unit. When applications allocate memory using the malloc C/C++ function call, the allocated memory is virtually continuous but not necessarily physically continuous. The “VIDIOC\_REQBUF” ioctl command is used to request the driver to allocate buffers, and the “VIDIOC\_QUERYBUF” is used to acquire various information about the newly allocated buffers such as size and index.

## v4l2 Buffer Passing Procedure

```
while(cond) {
    ioctl(v4l2_input_fd, VIDIOC_DQBUF, &buf);
    bufPtr = mmap(NULL,                               // "start" address (usually 0)
                  buf.length,                          // length
                  PROT_READ | PROT_WRITE,             // allowed use for map'd memory
                  MAP_SHARED,                          // allow sharing of mapped bufs
                  v4l2_input_fd,                      // driver/file descriptor
                  buf.m.offset);                      // offset requested from "start"
    doSomething(bufPtr, buf.length, ...);
    munmap(bufPtr, buf.length);
    ioctl(v4l2_input_fd, VIDIOC_QBUF, &buf);
}
```

- ◆ A simple flow would be: (1) DQBUF the buffer, (2) map it into user space, (3) use the buffer, (4) unmap it, (5) put it back on the driver's queue
- ◆ More efficient code would map each driver buffer once during initialization, instead of mapping and unmapping within the loop
- ◆ Alternatively, later versions of the driver allow you to create the buffer in 'user' space and pass it to the driver

This code section outlines the procedure for dequeuing a buffer, mapping it into the user's memory space, consuming the buffer, then unmapping the buffer and passing ownership of the buffer back to the driver using an enqueue operation.

While working code, this is not the most efficient means of accessing driver buffers because it requires a mmap and unmap operation each iteration of the loop. While mapping and unmapping the buffer is still more efficient than copying the entire buffer, it consumes unnecessary cycles. In most real systems the mapping of the buffers is brought outside of the loop, and the pointers returned by the mmap call are stored in a pointer array using the buffer index. The method shown above was used because it is a little simpler for a new user to quickly digest.

# Linux FBdev Display Driver

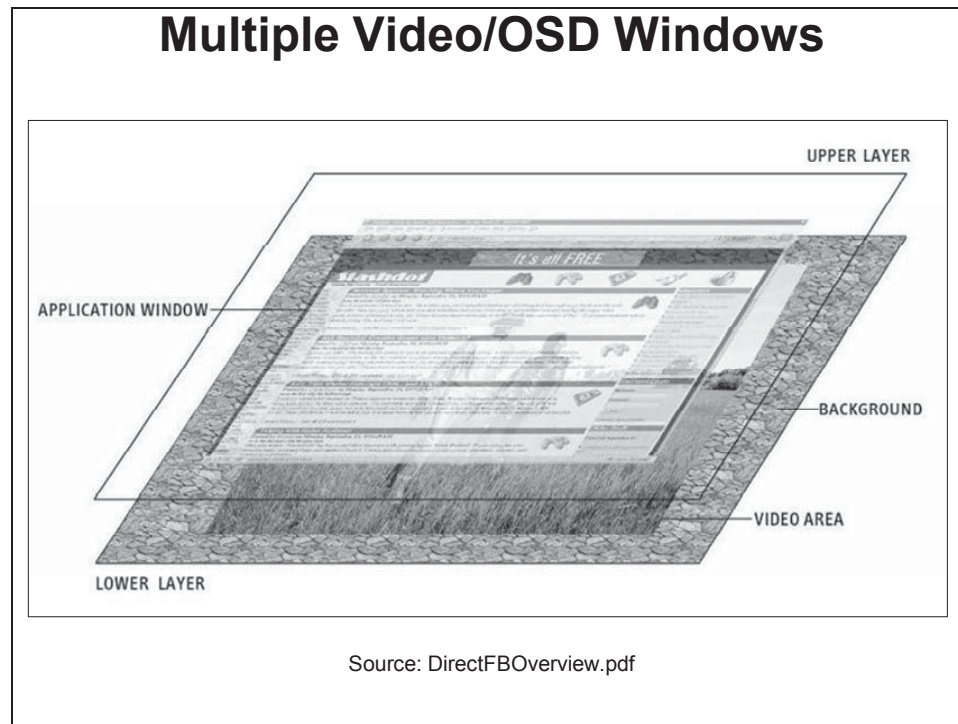
## FBdev Driver Overview

- ◆ **FBdev is a standard Linux video output driver used in many linux systems**
- ◆ **Can be used to map the frame buffer of a display device into user space**
- ◆ **Device nodes have major number 29**
- ◆ **Device nodes have a minor number x**
- ◆ **Uses /dev/fb/x node naming convention**

The Framebuffer Device (fbdev) Linux driver is used to display graphics. It generally uses a statically assigned major number of 29 and device nodes of /dev/fb/X, with the integer X corresponding to the different windows or layers supported.

fbdev is generally used to display graphics as opposed to v4l2, which is generally used to display video. Here, the term “graphics” is being used to refer to relatively low-motion displays that are composited by a computer, as opposed to “video” which describes high-motion content such as video stored in a compressed file. For instance, the “desktop” that a user sees when logging into a high-level O/S such as Linux would be considered graphics, as opposed to a downloaded H.264 file, which would be considered video.

Because graphics is relatively low-motion, the driver does not natively support multi-buffering, although it is possible to implement multi-buffering as will be shown. The fbdev driver is relatively simple, exposing a framebuffer, which is basically a pixel-by-pixel representation of the display in array format as a bitmap.



The fbdev driver supports multiple layers. In the context of the desktop environment that it was developed for, these layers would generally be used to support multiple application windows. The driver composites the multiple overlaid windows into the final output display.



## 32-bit Color Format

- ◆ Allows Pixel by Pixel Blending of OSD and Video Windows
- ◆ Uses an 8-bit, Attribute (Alpha Blending) Value

32-bit ARGB  
Pixel Value

| 8-bit<br>Attribute | 8-bit<br>Red | 8-bit<br>Green | 8-bit<br>Blue |
|--------------------|--------------|----------------|---------------|
|--------------------|--------------|----------------|---------------|

### 8-bit blending

0x00: 00.0%, 100% Video

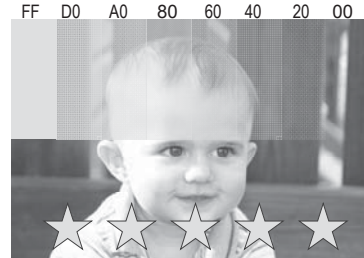
0x20: 12.5%, 87.5% Video

0x40: 25.0%, 75.0% Video

...

0xA0: 75.0%, 25.0% Video

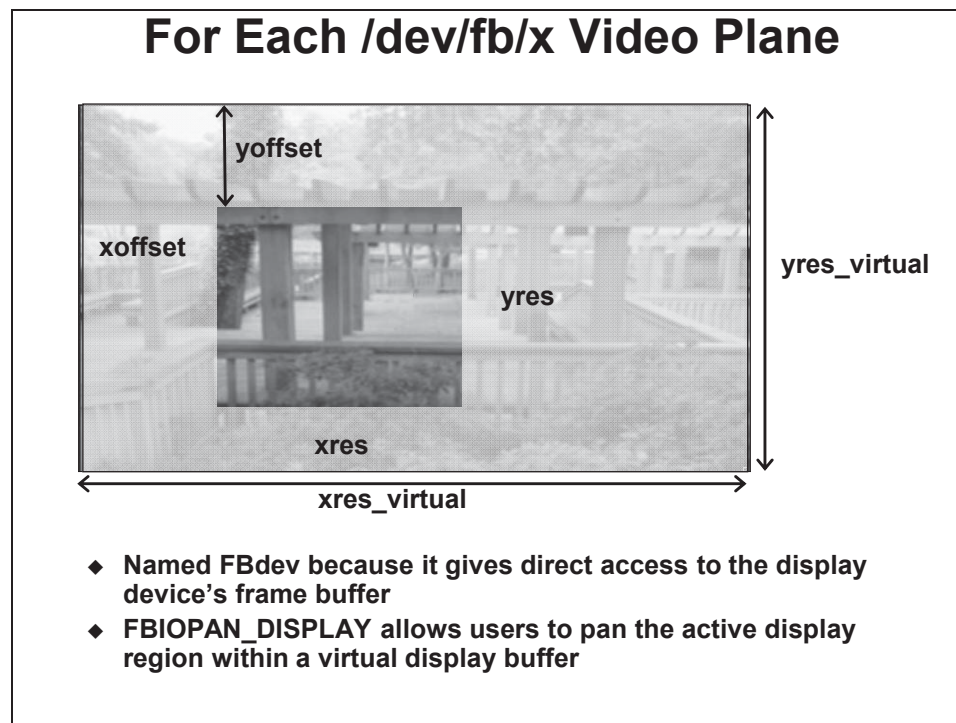
0xFF: 100%, 00.0% Video



The fbdev driver as implemented throughout the Linux community supports a number of different color spaces. The Arago driver for AM335x (Cortex-A8) devices, however, currently only supports the 32-bit ARGB color space. If the user attempts to configure the driver for a different color space, the driver will return a “not supported” error.

The format of the 32-bit ARGB color space is a packed 4-byte structure which has an attribute byte followed by red, green and blue bytes. The red, green and blue define a 24-bit color. The attribute byte defines a transparency, also known as an “alpha blending” parameter. When the attribute byte is set to all ones (0xFF) the window is completely opaque and shows none of the contents behind it. When the attribute byte is set to zero, the window is completely transparent, and only the background is displayed. Using other values allows the user to define 256 different levels of transparency.

Note that the attribute bit is only used on device that have hardware alpha blending support such as the DM6446 and the OMAP3530. Devices that do not have hardware alpha blending support, such as the AM3358 used in this workshop, ignore the attribute field, effectively setting it to 0xFF.

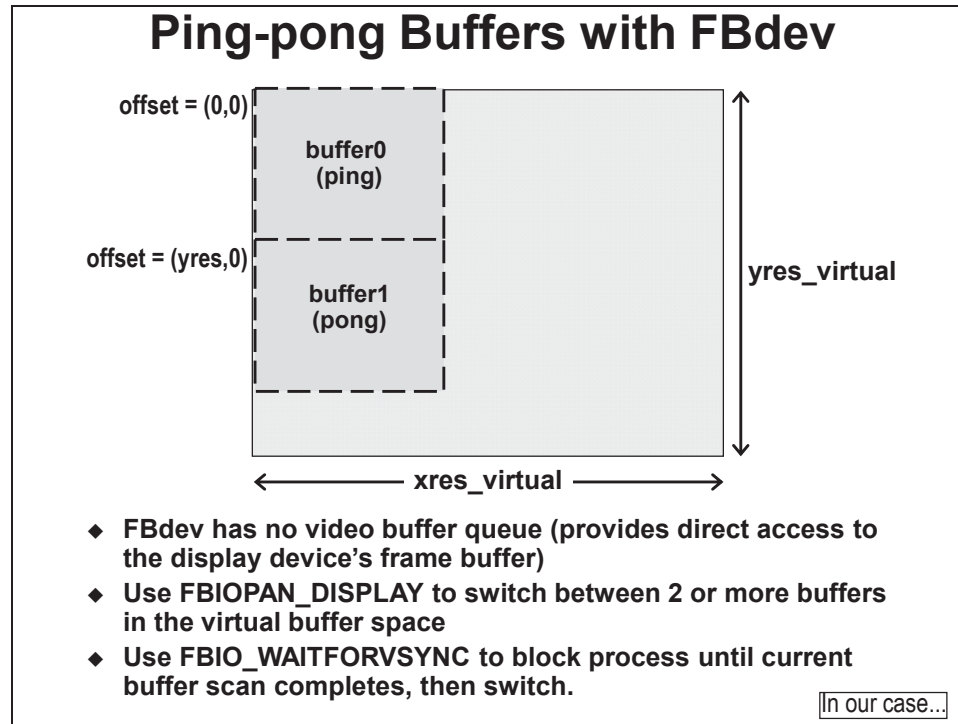


The fbdev driver supports multiple video planes. These planes may be resized and overlaid on top of one another to form the display. These planes are used by desktop managers to form application windows, which is their primary purpose. Embedded applications do not typically have desktop-styled displays with multiple application windows, but embedded developers may still use the fbdev planes for various purposes. Note that individual planes enumerated as “X” have the device node “/dev/fb/X” so that applications may write to separate video planes independently as their own virtual display.

The video planes have a virtual resolution and a display resolution. The virtual window, defined by `xres_virtual` and `yres_virtual`, is the display space that an application will write into. The display resolution, defined by `xres` and `yres`, is the portion of the virtual window that is actually displayed to be seen by the user. The driver also allows an x and y offset, which offsets the display space within the virtual space.

The concept of virtual and display spaces may make more sense with an example. Recalling that the driver is optimized for displaying application windows in a desktop environment, think of the video plane as an application window. The virtual space is defined by the native resolution of the application, and regardless of how the user modifies the application window, this virtual space will be unchanged.

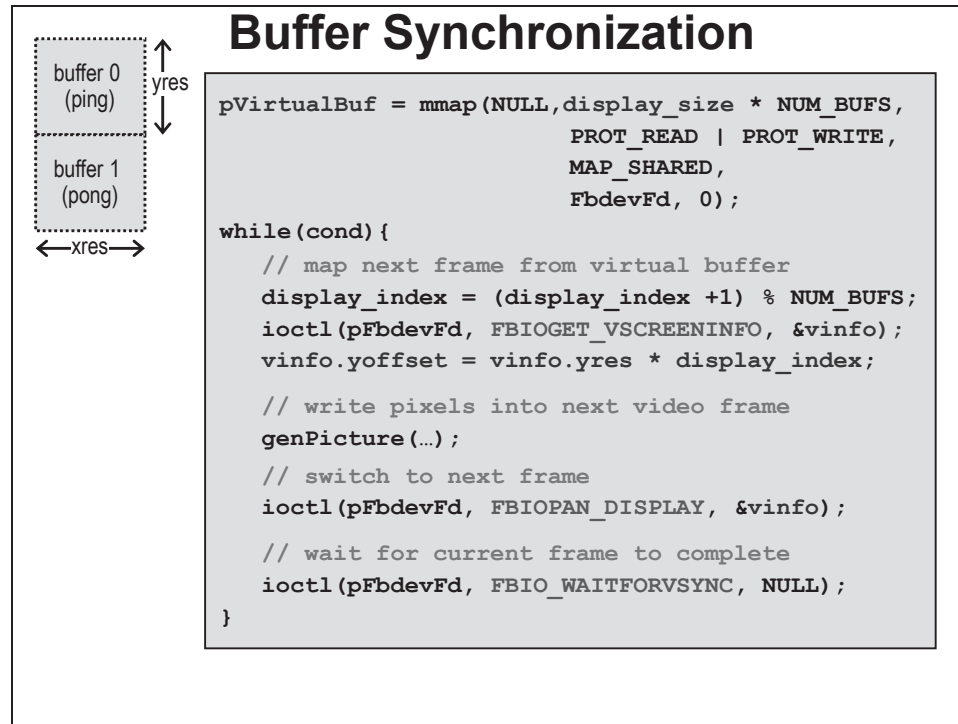
If the application window is resized, this will adjust the `xres` and `yres`, which is the size of the display area. When this is a smaller size than the application output, scroll bars will appear in the window. When the user moves the scroll bars, this will adjust the x and y offsets accordingly. As the display window is resized, scrolled or moved, the virtual space remains unchanged. This decouples the application from the display window, which simplifies the application interface and improves performance.



Since fbdev was developed for graphical displays, which, at least traditionally, have a relatively slow rate of change, the driver was not implemented with multiple buffers as is the v4l2 driver. Instead, the driver presents a memory array, the “framebuffer” and there is a 1:1 between the pixel values in the array and the physical pixels on the display, so the driver is designed to display the same buffer that is being updated by the application.

If the displayed graphics have a high degree of motion, this may cause motion artifacts on the display. This may be solved by introducing a ping-pong buffer scheme. Unlike v4l2, there is no native support for multibuffering in the driver, but it is possible for a user to implement ping-pong buffers by hand by taking advantage of the virtual display space.

The basic implementation is to create a virtual space that has a resolution that is N times as large as the display space, where N is the number of buffers supported. The FBIOPAN\_DISPLAY ioctl command is then used to jump between the buffers. This is in lieu of the panning ability of scrollbars for which the command was designed.



If a ping-pong buffer scheme is used with fbdev, then a mechanism must be used to ensure that the updating of the offset using FBIOPAN\_DISPLAY is executed in sync with the vertical synchronization signal of the hardware display. The vertical sync signal is a hardware signal that is generated after a full frame has been placed on the display.

The FBIO\_WAITFORVSYNC ioctl command is a blocking call that will block until a vertical sync is received from the display. Note that in the code above, the display is first panned and then the wait for vsync is called. This seems backwards. It would be more intuitive to wait for the vertical sync and then update the display to the new frame.

The reason that the function calls are made in this order is as follows: the FBIOPAN\_DISPLAY call is actually hardware latched. This means that, even though the application will return from the function call immediately, the panning operation that updates the offset will not actually take effect until the vertical sync is received. This is implemented via shadow registers that are used to update the offset registers in hardware by the hardware. This guarantees that the panning operation will be performed at the correct point, as the window for the panning to be performed without creating a “tearing” artifact is narrow.

Why then is the FBIO\_WAITFORVSYNC function needed? It is required because the FBIOPAN\_DISPLAY call returns immediately and the application continues. Without the wait for vsync, the application may get off synchronization with the driver, so the wait for vsync call is required to synchronize the application, even though the synchronization of the pan display operation is handled in hardware.

## Blocking on select()

```
int wait_for_frame(int fd) {
    struct timeval tv;           // timeout value
    fd_set      fds;           // set of file descriptors to wait on

    FD_ZERO(&fds);              // using a file descriptor set of only one
    FD_SET(fd, &fds);           // value – passed in as function parameter

    tv.tv_sec = 2;              // timeout after 2 seconds if no new frame

    // Use select to wait for next filled frame
    ret = select(fd + 1,        // largest file descr value to test
                 &fds,          // set of fd's to wait on "read"
                 NULL, NULL,    // fd's to wait on write or error
                 &tv);          // timeout value

    ...
}
```

### ◆ Reasons to use select():

- ◆ Can block on multiple events
- ◆ Allows specification of a timeout value for blocking

What if a hardware issue caused the display to lock up so that a vertical sync signal was never received? Then the wait for vsync operation would be blocked forever, and the application would never continue. This may not be the desired behavior of the blocked thread is responsible for more than just generating the display.

For this reason, some programmers choose to implement timeouts on some of their blocking function calls. A timeout will unblock after a specified period of time even if the semaphore that is being waited on never posts. This allows the programmer to recognize that an issue has occurred and have the application respond appropriately. For instance, for a 30 Hz display, a programmer might set a 1 second timeout. If this timeout occurs, then clearly the display is not operating correctly since it should generate a vertical sync 30 times each second.

Most of the standard blocking calls in Linux do not have an integrated timeout; however, the “select” function may be used with any file descriptor to block with a timeout. The select function is so named because it has the additional functionality of allowing the user to specify an array of file descriptors and block until any one of the file descriptors in the array becomes unblocked or until the timeout is reached. If the user wants to block on a single file descriptor with a timeout, then the select call may still be used, but the file descriptor should be specified as a file-descriptor array of size one element as shown in the example above.

## Commonly Used FBdev ioctl

### Data Structures

```
struct fb_fix_screeninfo myFixScreenInfo;  
myFixScreenInfo.smem_len;    // length of framebuffer  
  
struct fb_var_screeninfo myVarScreenInfo;  
myVarScreenInfo.xres;        // visible pic resolution  
myVarScreenInfo.xres_virtual; // virtual pic resolution  
myVarScreenInfo.xoffset;     // from virtual to vis
```

### Get or put variable screen information

```
ioctl(fd, FBIOGET_VSCREENINFO, &myVarScreenInfo);  
ioctl(fd, FBIOPUT_VSCREENINFO, &myVarScreenInfo);
```

### Get fixed screen information

```
ioctl(fd, FBIOGET_FSCREENINFO, &myFixScreenInfo);
```

### We use Pan to switch output buffers

```
ioctl(fd, FBIOPAN_DISPLAY, &myVarScreenInfo);
```

### After writing buffer and pan\_display, wait for current to finish

```
ioctl(fd, FBIO_WAITFORVSYNC, NULL); // arg 3 is not used
```

These are some of the most commonly used structure elements and function calls for the fbdev driver. The “vscreeninfo” structure holds variable parameters, i.e. those parameters that may be set in software by the application. As such there is a get and a put operation for the vscreeninfo. The “fscreeninfo” structure holds fixed parameters, generally those parameters that are fixed by the hardware and can only be changed by a recompiling of the driver. Thus there is only a get operation for the fscreeninfo.

## Lab Exercise

### Lab 9

**Lab 9a: Copy-based Framebuffer Application**

**Lab 9b: mmap-based Framebuffer Application**

**Lab 9c: Double-buffered Application**

(Page intentionally left blank)



# Module 10: Qtopia GUI Development

---

## Introduction

The previous module introduced FBDEV, the Linux device driver that is commonly used for the display of graphics. This driver, however, is very low level, with the display being specified in the form of a bitmap.

Typically a developer would prefer to have additional tools such as fonts for displaying text and an abstraction of various common GUI components such as text and input boxes. There are many packages in Linux that may be used for the development of a graphical user interface. One of these packages that is commonly used in the embedded space is called Qtopia (or “qt” for short.)

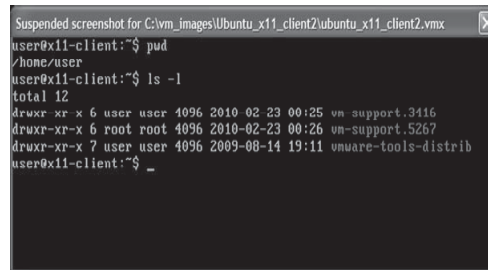
Qtopia provides abstractions for many commonly used GUI components (termed “widgets”) as well as providing font support and human interface device (HID) support to integrate mouse and keyboard inputs as well as touch-screen inputs. Furthermore, Qtopia has a feature-rich and easy to use development environment called QT creator.

## Module Topics

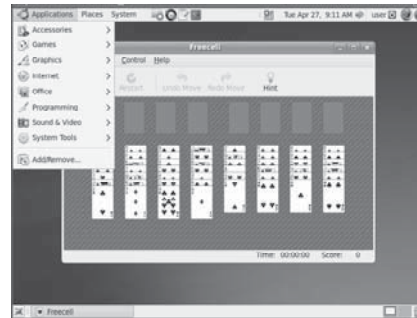
|   |              |
|---|--------------|
| <b>Module 10: Qtopia GUI Development.....</b> | <b>10-1</b>  |
| <i>Module Topics.....</i>                     | <i>10-2</i>  |
| <i>Graphics Software Layers.....</i>          | <i>10-3</i>  |
| <i>Qtopia .....</i>                           | <i>10-8</i>  |
| <i>Lab .....</i>                              | <i>10-19</i> |

# Graphics Software Layers

## Terminal vs. Desktop



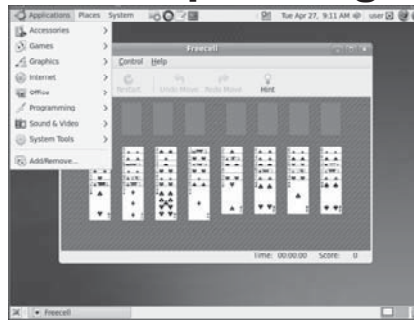
```
Suspended screenshot for C:\vm_images\Ubuntu_x11_client2\ubuntu_x11_client2.vmx
user@x11-client:~$ pwd
/home/user
user@x11-client:~$ ls -l
total 12
drwxr-xr-x 6 user user 4096 2010-02-23 00:25 vm-support.3416
drwxr-xr-x 6 root root 4096 2010-02-23 00:26 vm-support.5267
drwxr-xr-x 7 user user 4096 2009-08-14 19:11 vmware-tools-distrib
user@x11-client:~$ _
```



- ◆ A terminal is a text-based interface to Linux O/S
- ◆ Commands are interpreted by a shell program (such as BASH)
- ◆ A terminal may connect via RS-232, Ethernet, etc.
- ◆ xterm is a terminal which connects via the X window server, i.e. receives input via x11 HID and outputs to x11 display
- ◆ Most applications which can be launched from the Desktop (i.e. a drop-down menu) can be equivalently launched by typing a command into a Linux terminal

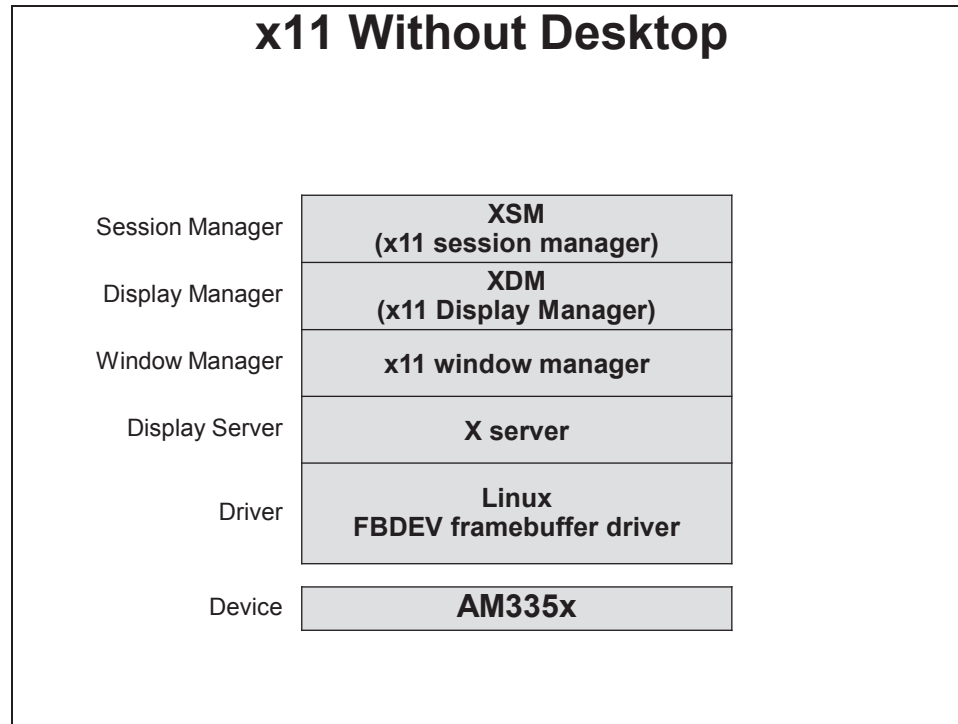
Linux is a fundamentally terminal-based operating system, meaning that the primary form of interaction between the user and the O/S is handled via the terminal. This is not to say that users who prefer a GUI interaction environment do not have this option. However the operating system may be fully accessed via the terminal and is therefore capable of operating in a text-only environment. This is one of the reasons that Linux is a very popular choice of operating system on servers, which are often accessed remotely via text-based protocols such as secure shell.

## Desktop Managers



- ◆ Gnome and KDE are two of many Linux display managers
- ◆ Desktop managers provide a “Desktop” or similar environment (i.e. for handheld device)
- ◆ Applications (such as mplayer or xterm) typically interface directly to x11 and are therefore independent of the Display Manager
- ◆ Gnome display manager is “just another client” to x11
- ◆ A display manager isn’t required to run Linux
- ◆ x11 isn’t required to run Linux, though it is required by many graphical applications

The graphics layers that eventually stack to a user desktop are not integrated into the Linux kernel, but are instead software layers that stack on top of it. For this reason, there are actually a number of graphical environments that users may choose from, including XORG, qtopia and the K desktop.



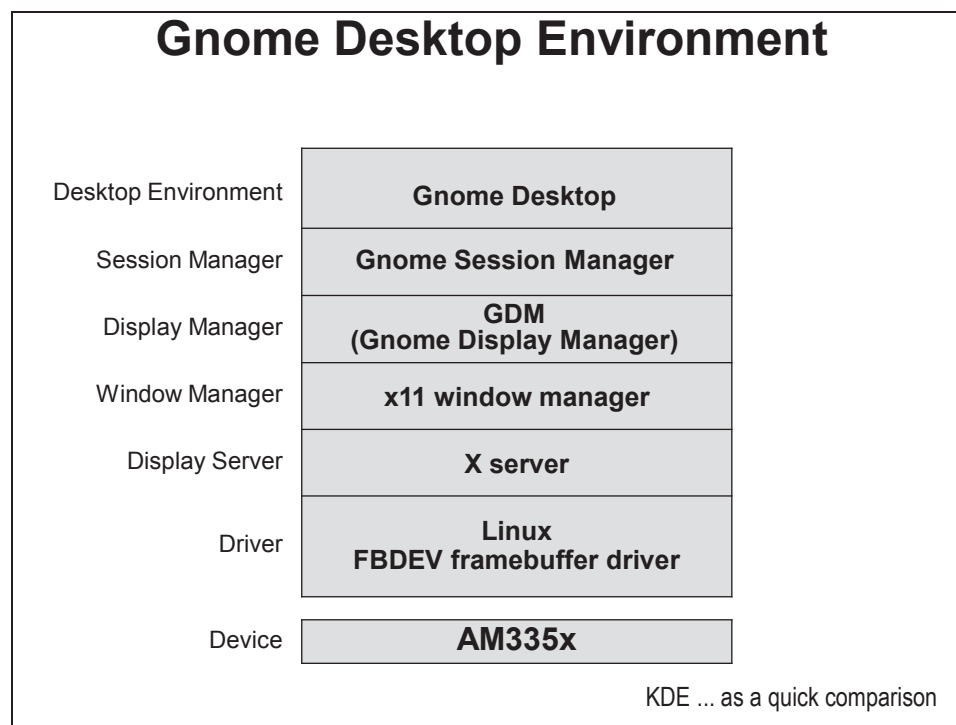
XORG was one of the first graphics protocols developed for Linux, developed in the 80's, and is only now, 30 years later, beginning to be phased out of many Linux distributions in favor of Wayland.

One of the reasons for the longevity of XORG is undoubtedly the network transparency that was built into the graphics stack at its most fundamental layer. The X server sits directly on top of the fbdev graphics driver and provides a network-transparent access to the driver. In other words, and Xorg client on the network, assuming it has the proper permissions, may access the fbdev driver. This makes running a remote desktop trivial on an Xorg system, the user need only set the DISPLAY environment variable, and the desktop is exported to another computer on the network.

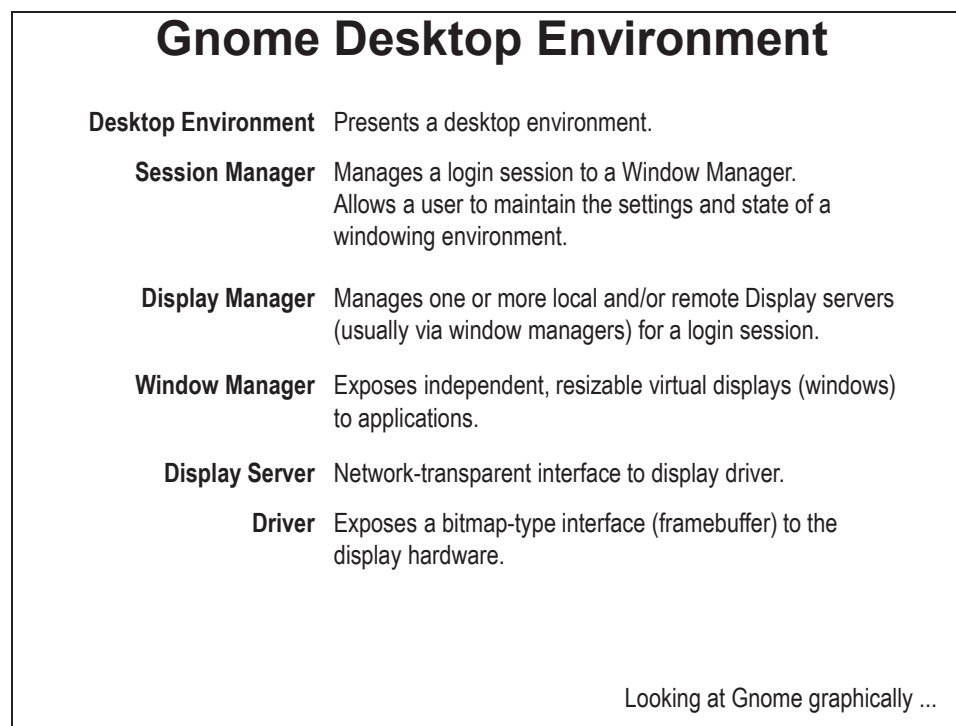
Above the xserver sits the window manager, which interacts with applications to provide a common window interface to each application, as well as an interface to the user to resize and move the windows about the display.

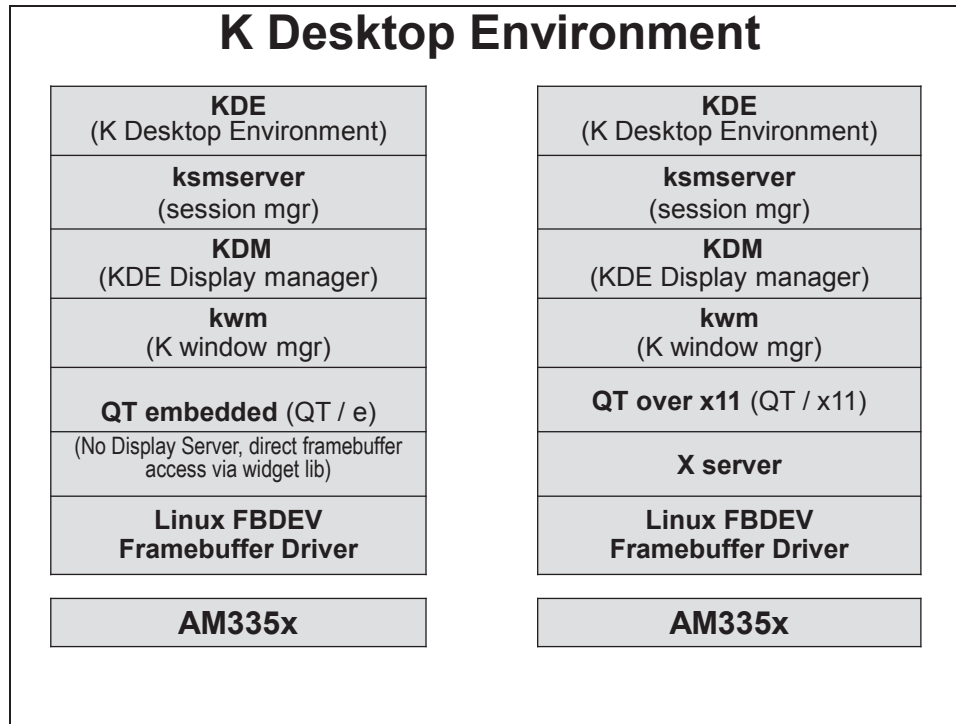
The display manager is the next layer and provides a large number of fonts, colors, etc., that may be used to customize the “look and feel” of the display. The layers below have font support; however, the user is left with a small number of fonts and color with little ability to customize.

The session manager is the topmost layer of the XORG stack, which does not have desktop support natively. The session manager manages user login sessions and works closely with the display manager to save and restore display configurations for each user that initiates a session, allowing each user to customize the display configuration according to their individual taste.



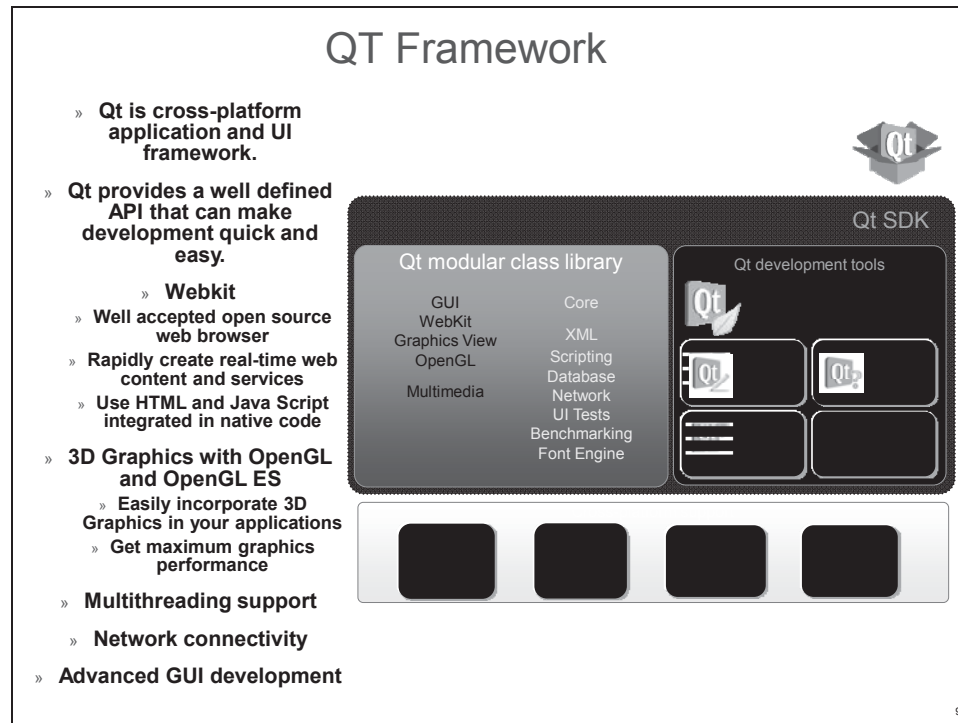
If the user wants a desktop environment, there are many to choose from. One popular desktop, Gnome, was developed to overlay the Xorg stack. It utilizes the X server and x11 window manager directly from the Xorg stack; however, the Gnome stack uses its own display manager which allows the Gnome desktop to display a different look and feel than that of the standard Xorg stack. Then, since the session manager sits over the display manager, Gnome implements its own session manager as well.





Another popular desktop environment is the K desktop. Whereas the Gnome desktop was developed for use with the Xorg stack, the K desktop was developed over a Qtopia layer. Qtopia is a GUI development platform that will be discussed further later in this module. One disadvantage of X11, particularly in an embedded environment, is that it is somewhat heavy in terms of its usage of memory and CPU cycles. Qtopia for embedded (QT/e) is very popular for embedded development because it uses significantly fewer resources. Interestingly, Qtopia can also be run over X11, allowing K to run either on a native embedded Qtopia stack or over X11 using QT/x11. This gives developers a very powerful debugging tool as they may develop and debug a GUI interface on a desktop platform and then very easily port to their embedded platform after debugging has been completed.

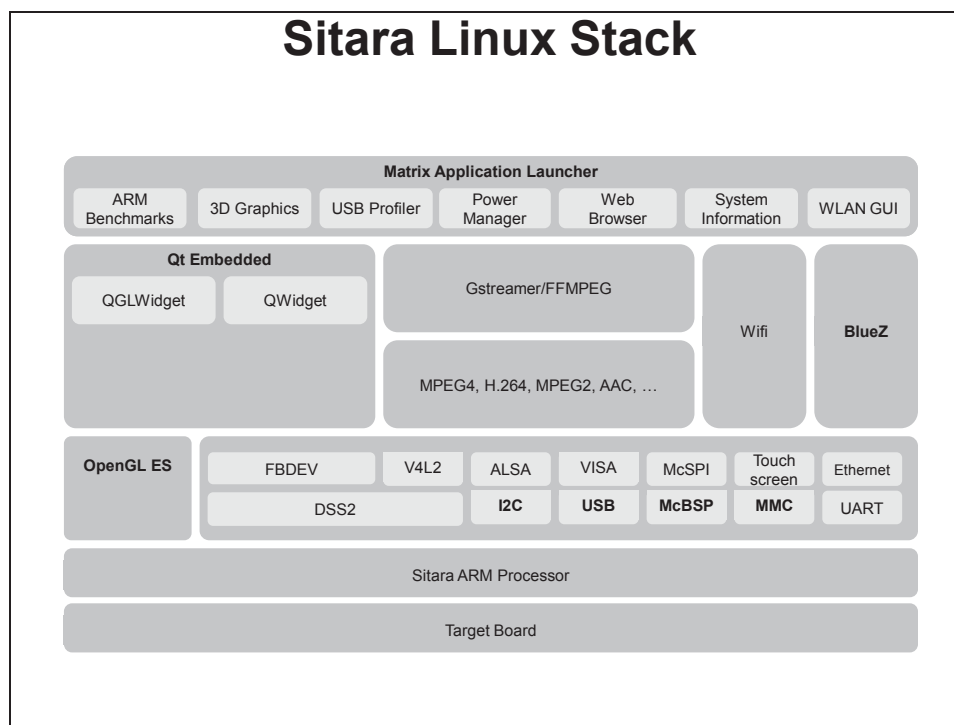
# Qtopia



Qtopia is one of the most popular GUI development tools in the Linux community, particularly for embedded applications. One important feature of QT is that it is supported on Windows and Linux desktops as well as embedded Linux systems, allowing the user to develop on a variety of platforms and easily port between them.







This diagram shows some of the major constituent software packages in the Arago distribution, particularly with regards to graphics and video. For graphics development, TI recommends the QT Embedded package, which sits over the fbdev driver and the OpenGL ES library, which is accelerated by the SGX530 2D/3D graphics accelerator that is built into the AM335x devices.

For applications that require video support, the Gstreamer framework is recommended. Gstreamer has the capability to decode or encode a rich variety of formats and utilizes the FFMPEG codec package.

Also, TI provides Wifi and Bluetooth support integrated into the Arago distribution using TI's Wilink products.

## “Hello World!”



```
#include <QApplication>
#include <QLabel>

int main(int argc, char **argv)
{
    QApplication app(argc, argv);

    QLabel label("Hello World!");
    label.show();

    return app.exec();
}
```

Here is an example Qtopia application written in C++. All Qtopia applications utilize the QApplication library and header file and contain a QApplication object, whose exec method is the final call made in the main thread. Note that this function does not return until the Qtopia app is closed, so it should be the final call in main.

The application then includes the library and header file for any widgets that it will use. In this example the QLabel widget is initialized with the “Hello World!” string. QT widgets are initialized hidden by default, so it is required to call the show method for them to be made visible on the display.

The result of the application is a message box as shown at the top. Note that the window may be resized, minimized or closed using the attached human interface device. All of the necessary support is already built into the widget.

## Running “Hello World!”

- Run qmake inside the helloworld directory to create a project file
  - `qmake -project`
- Run qmake again to create a Makefile from helloworld.pro
  - `qmake`
- Run make to build the application
  - `make`
- Application is built and ready in debug/ directory. Copy executable to your filesystem on your target and run.

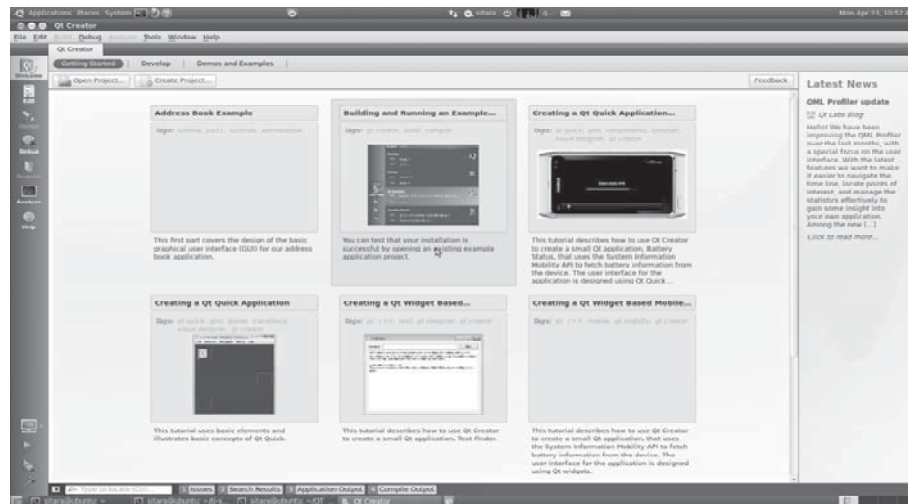
The basic procedure for building a QT project is shown above. The first step is to create a project file. “qmake,” which is the qt build application, will create a project file from all source files in the current directory if called with the “-project” option.

qmake does not actually build the application itself; it utilizes the makefile utility. Calling qmake in a directory that has a project file will build the corresponding makefile for the project. Once this is completed, simply call make to build the application.

## QT Creator – Development tools

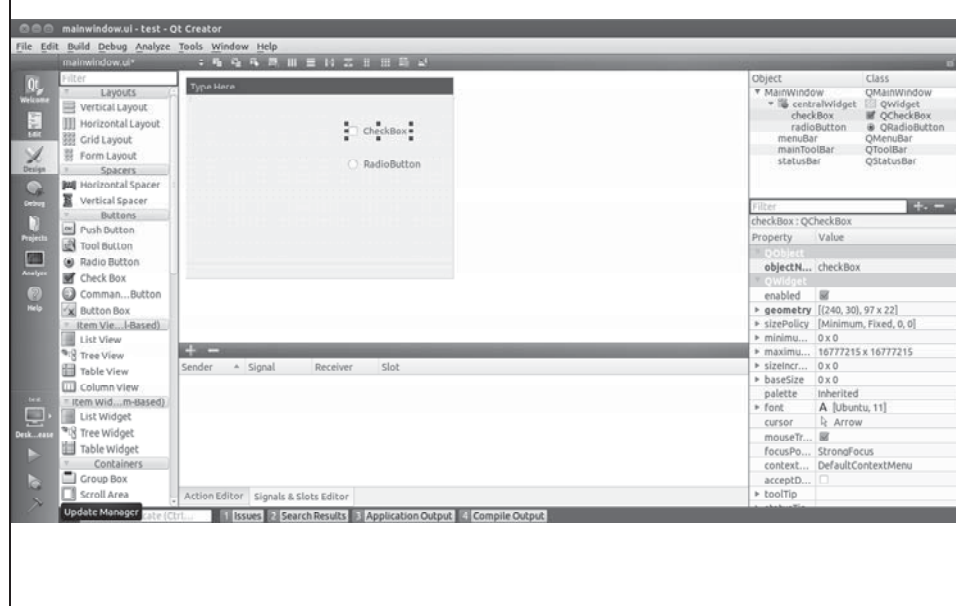
- Downloading and setting up:

[http://processors.wiki.ti.com/index.php/Qt\\_Creator\\_Embedded\\_Debugging\\_Setup](http://processors.wiki.ti.com/index.php/Qt_Creator_Embedded_Debugging_Setup)



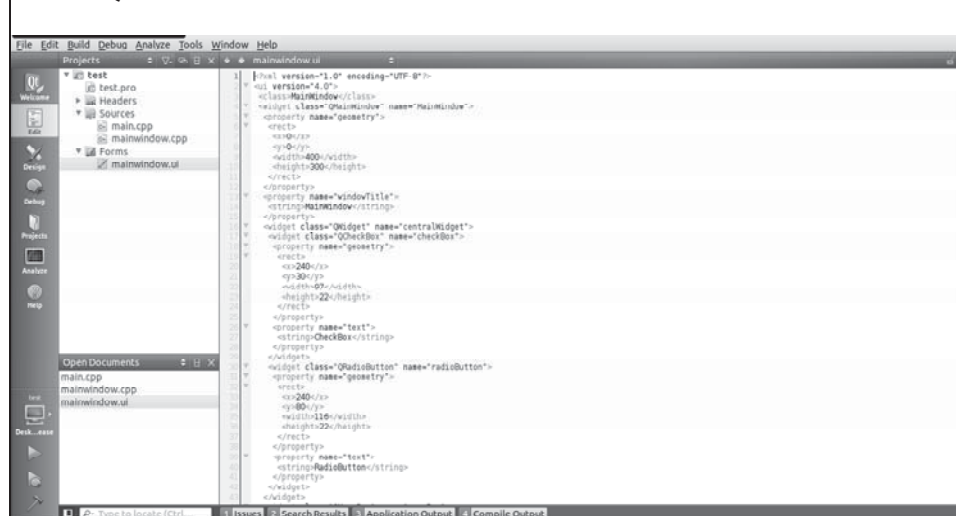
QT also has a GUI development tool named QT Creator. Qt Creator is an integrated development environment with all of the tools needed to generate, build and debug user interface applications.

## QT Creator – Drag and Drop Widgets (Forms)



One of the most powerful features of Qt Creator is the forms view. This is a “What You See Is What You Get” editor that allows developers to drag widgets from a pane into a virtual display area. The resolution of the virtual display can be set as desired, and the widgets are easily resized, labelled and otherwise modified.

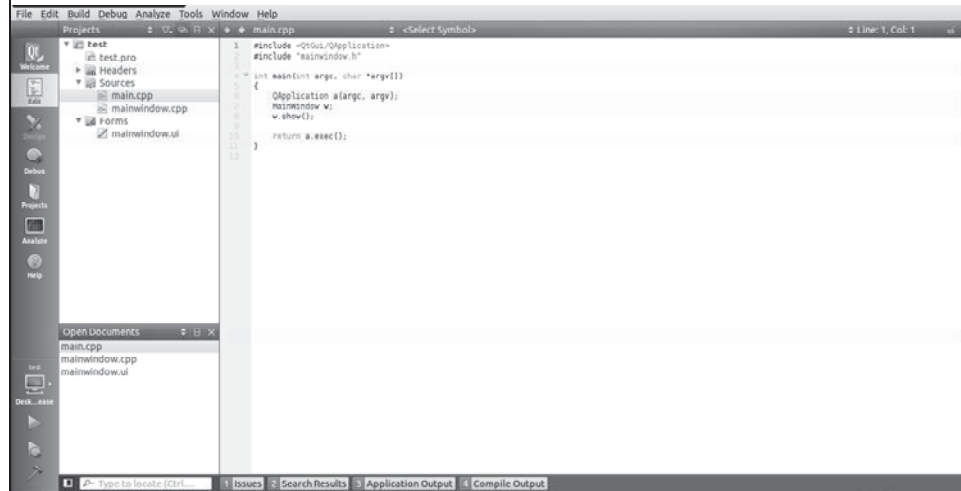
## QT Creator – Forms Stored via XML



The UI created on the previous slide is stored as .xml  
 QT Creator then generates the UI from this .xml file

The output of the forms view editor is an XML file

## QT Creator – C++ Objects

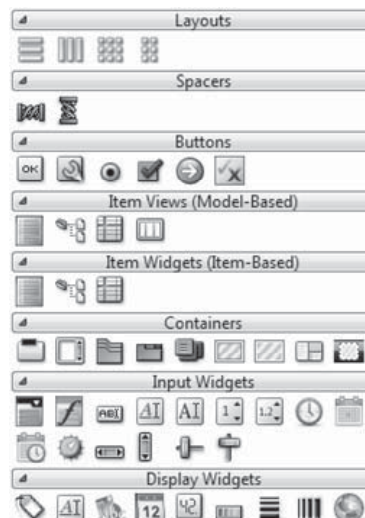


Alternately, users may program using C++ object classes for each widget

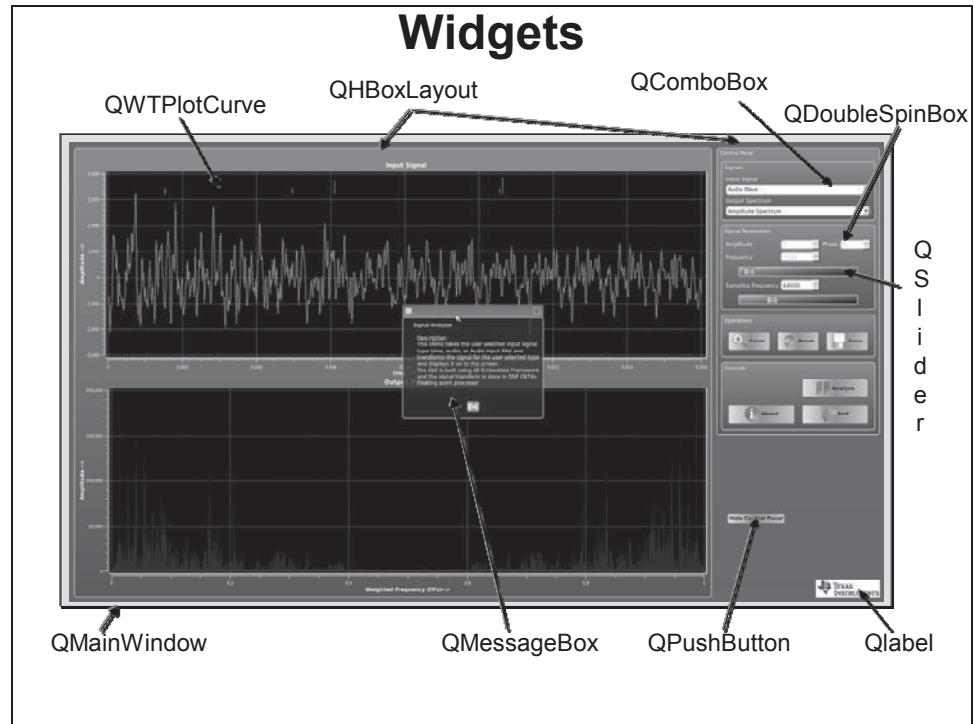
In order to build an application with the form-view generated XML file, simply add the “Mainwindow” library into the QT application, declare a Mainwindow object and call its show method. QT creator generates a Mainwindow.cpp file from form-view XML file and links it into the final application.

## Widgets

- ◆ Qt UI framework is based on widgets
- ◆ Widgets respond to UI events (key presses/mouse movements), and update their screen area
- ◆ Each widget has a parent, that affects its behavior, and is embedded into it
- ◆ Most Qt classes are derived from QWidget
- ◆ Refer to online documentation at
  - <http://doc.qt.nokia.com/4.6/qwidget.html>
  - Tip – Documentation is arranged using class names.



Widgets is QT’s name for the C++ objects that it defines to handle various UI tasks.



This demo UI was created entirely within QT. The diagram shows each of the widget types that were used. The FFT function call that is used to translate the time-based signal into the frequency domain is not a part of Qtopia. This function is tied into the user interface using the signal and slot mechanism.

## Signals & Slots

- ◆ **Signals & Slots**
  - ◆ **Signals and slots are used for communication between objects. The signals and slots mechanism is a central feature of Qt and probably the part that differs most from the features provided by other frameworks.**
- ◆ **Signals**
  - ◆ **Events occur and cause a signal**
    - ◆ **Widgets contain a list of predefined signals, but you can subclass a widget to add your own signal**
    - ◆ **Example – button press, or Process complete**
  - ◆ **Slots are the functions which are assigned to handle a signal.**
    - ◆ **Widgets contain a list of predefined slots.**
    - ◆ **You can subclass a widget and add your own slots**

Signals and slots are the basic interface between the QT widgets and the underlying application. Signals are event driven, generally raised when the user interfaces to one of the input widgets on the UI. Signals are plugged with a callback function so that when the signal is raised, QT calls the callback function in response.

Slots are the interface to the QT display widgets. Slots are simply functions that, when called, change the output of the display widget. These functions take one or more parameters that manipulate the output of the display widget.



## Simple Signal/Slot Example

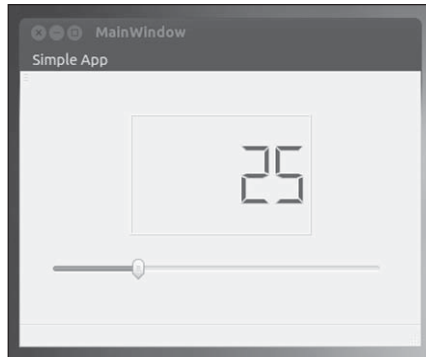
The signal “sliderMoved(int)” is generated whenever the horizontal slider is moved

By adding this signal to the “lcdNumber” “display(int)” slot, the LCD will display the value of the horizontal slider when it is moved.

The value of an input widget may be tied to the output of a display widget by registering the slot function of the display widget as the callback function of the input widget. In the example above, the slider widget has a signal named “sliderMoved.” When a signal is generated, an input widget will call a callback function, passing its captured value. The “(int)” designation of the sliderMoved signal indicates that it will pass an integer to its callback function.

The lcd display has a slot named “display.” This is a C-callable function that takes a single integer argument as indicated by the “(int).” Within QT Creator one may tie the slot of a display widget to the signal of an input widget using the interface shown. In the generated C/C++ code, this is registering the “display” method of the lcd widget object as the callback function of the slider object.

## Simple Signal/Slot Example



Here a GUI object signal is tied directly to another GUI object slot; however, programmers can also write C++ functions to interface to signals and slots, thus connecting their interface to back-end functionality.

Once the QT application is launched, any movement of the slider object will generate a signal. Since the “display” method of the lcd widget object is registered as the callback function of the slider object, the display method will be called whenever this signal is generated, with the output of the slider object, which is an integer representation of its position, passed into the display widget.

The result, for this simple example is an lcd-style display that updates from 0-100 according to the position of the slider bar.

In this example, two QT widgets are connected via the signal/slot mechanism, but user functions may be attached to QT objects using the same paradigm. A user function may be registered as the callback function of the slider object, and any time that the slider is moved, that user function will be called. Likewise, the “display” method of the lcd widget object can be directly called from within any user function.

# Lab

## Lab Exercises

**Lab 10a: “Hello World” application on the x86**

Because of the portability of QT it is generally simpler to develop applications on the x86 host machine and later transfer to an embedded platform.

**Lab 10b: “Hello World” application on the AM335x****Lab 10c: Remote debug of “Hello World” application on the AM335x using gdb/gdbserver**

QT Creator has remote debug capability built in, similar to the debugging that you have been using within Code Composer Studio.

(Page intentionally left blank)

# Module 11: The git Tool

---

## Introduction

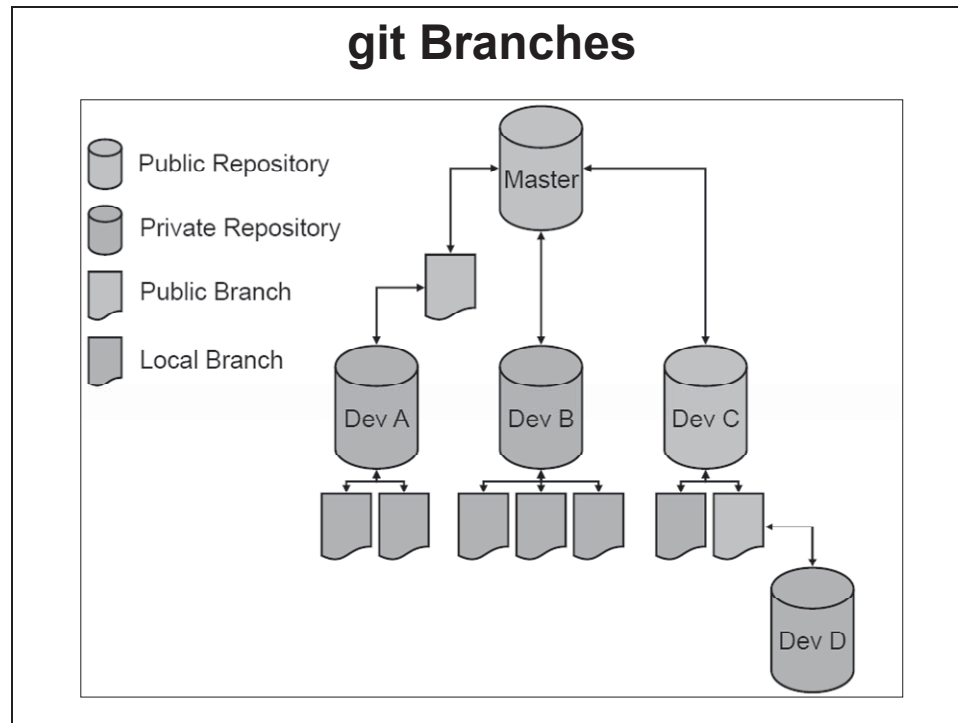
The git tool is an open-source software package used for managing different versions of software. git is used throughout the Linux community, and is the version management tool used by the developers of the Linux kernel itself. Furthermore, Arago, the free, open-source distribution that is tested and maintained by Texas Instruments, uses git as its versioning software.

By learning the git tool, you will be able to apply updates and patches from the Arago distribution with far less effort than would be required to do the same task manually. Furthermore, git has a plugin to Code Composer studio that will be discussed in the next chapter. This plugin allows you to use git to seamlessly manage any CCS project.

## Module Topics

|  |              |
|--|--------------|
| <b>Module 11: The git Tool.....</b>      | <b>11-1</b>  |
| <i>Module Topics.....</i>                | <i>11-2</i>  |
| <i>git Overview.....</i>                 | <i>11-3</i>  |
| <i>Adding Commits.....</i>               | <i>11-7</i>  |
| <i>Adding and Merging Branches .....</i> | <i>11-16</i> |
| <i>Lab 11: Using git.....</i>            | <i>11-24</i> |

## git Overview



Here is a top-level diagram of a basic git development structure. git is a distributed version management system, meaning that each developer “clones” the master git repository into a copy on their local machine. This allows developers to work independently, even when they do not have access to an internet connection.

The diagram shows public and private branches and repositories, where a repository is a warehouse of code containing one or more code branches. Public repositories and branches are available to be cloned onto a remote machine. Private repositories and branches are available only on the local machine, although they may be manually “pushed” upstream to a public repository. Note as shown on the diagram that a cloned repository may be configured as a public repository, allowing developer C to clone the master repository and then make this a public repository which developer D may clone.

## Creating a new git repository

### You may create a new git repository

```
host$ mkdir ~/git/my_project
host$ cd ~/git/my_project
host$ git init
host$ cp ~/project_files/* ~/git/my_project
host$ git add *
host$ git commit -m "initial_version"
```

Step 1: make a directory to hold your project files

Step 2: use “git init” to initialize the directory with git metadata

Step 3: add files to the project

Step 4: use “git add” to add files from the directory to the git repository

(just placing the file in the git directory hasn’t added them to get yet!)

Step 5: use “git commit” to commit the current state of the git repository,  
with the reference tag “initial\_version”

Creating an empty, local repository requires only a few steps. First, an empty directory is created where the repository will be located, and then the “git init” call is made to initialize the repository.

When the repository is first created, it has a single branch, named “master,” but no commit nodes on the branch. An empty repository behaves differently than the typical repository, which has one or more commit nodes, so it is generally advisable to also create or copy in a set of initial files, add them to the repository and create an initial commit node. More on the commit procedure will follow.



## Cloning an Existing git Repository

**By cloning a public repository, you create a local copy that can be accessed without internet connectivity.**

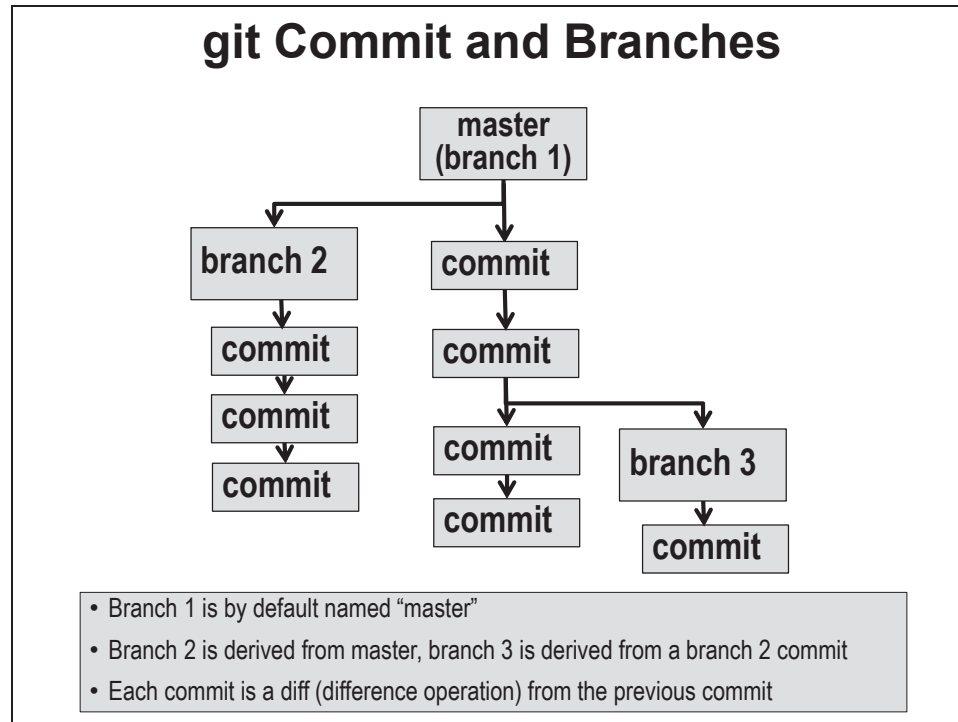
```
host$ mkdir ~/git
host$ cd ~/git
host$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Cloning is better than simply copying source files because

1. Automatically creates a local git repository
2. Allows you to pull external changes (patches) from the public repository and merge them into your copy
3. Depending on configuration, allows you to push your changes back to the public repository

Another means of create a git repository is to clone a pre-existing public repository. This is done by creating an empty directory and then calling “git clone” with the Uniform Resource Locator (URL) of the pre-existing repository.

In addition to copying the repository, “git clone” will configure the local repository to interface to the original repository via the “push” and “pull” commands, which are used to “push” updates from the local repository to the remote repository or “pull” updates from the remote repository to the local repository.

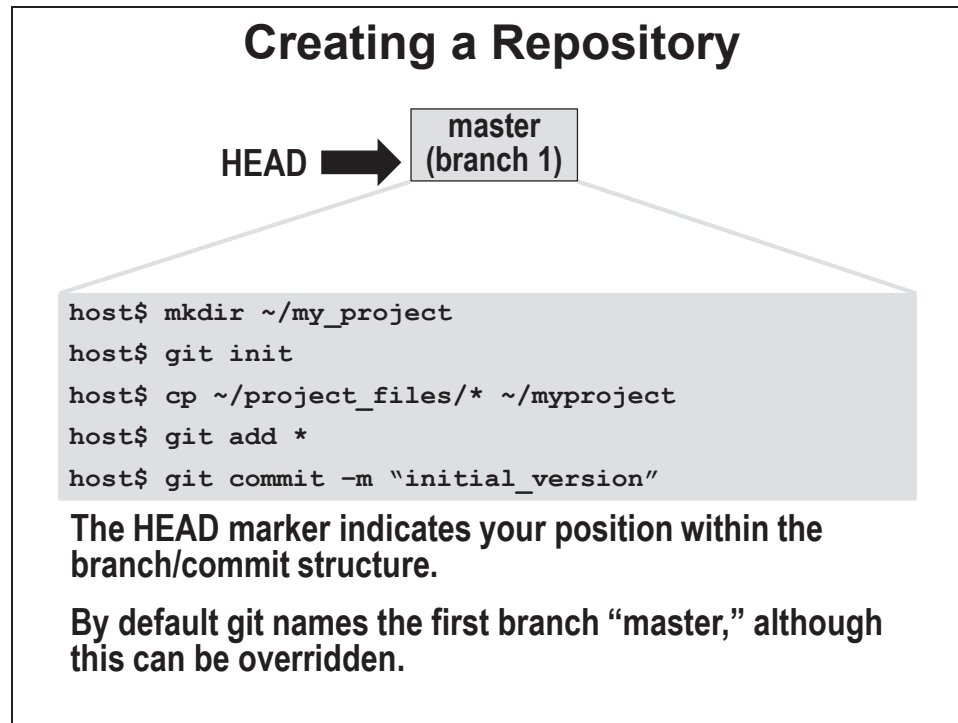


The basic elements that form a git repository, also referred to as a git “tree,” are commit nodes and branches. New repositories are initialized with a single branch, named “master.” At various points during code development, a user can create a commit node with the “git commit” command. Commit nodes are snapshots of the code at a given point in time that can be viewed or reverted back to at a later date.

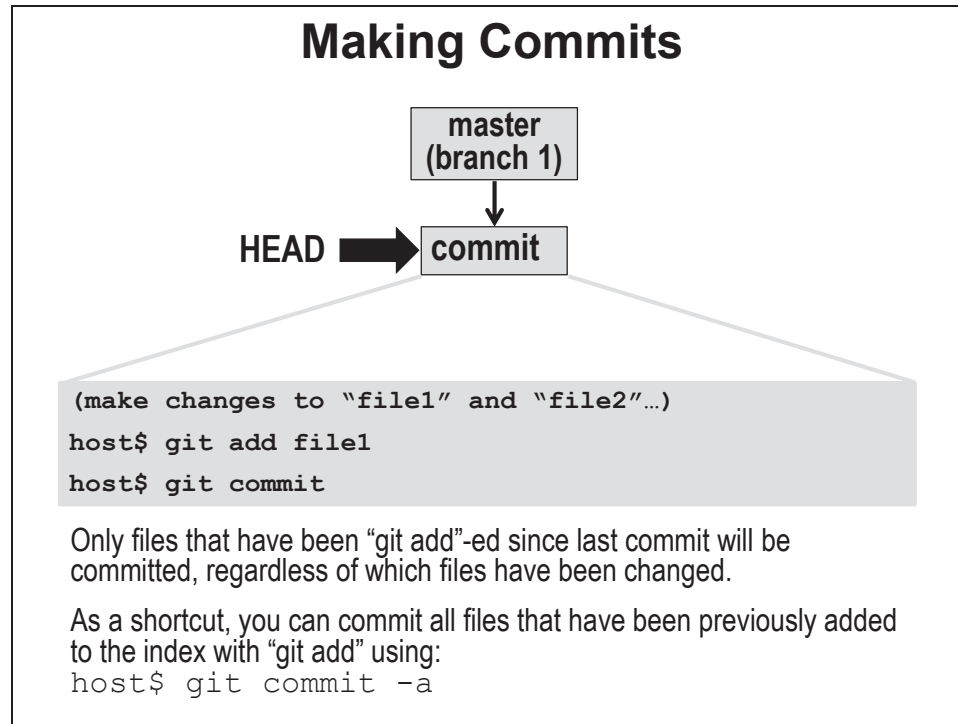
As commit nodes are sequentially made, they extend a single branch, which is basically a documenting of the evolution of the code over time. Additionally, users can create a new branch from any commit node on the tree. When a new branch is created, it provides a new pathway for the code to evolve without affecting the commit nodes of the original branch. Managing multiple branches then allows development of the code along different pathways that do not affect each other, providing various “sandboxes” for the code to be developed within.

For instance, if multiple developers are working on the same code base, they might each want to create their own branch so that they can modify code within the branch without affecting the code development of the other developers, who are each working within their own branches. Then at some point these developers may wish to combine their individual contributions together into a single code base, which may be done using the “git merge” command that will be discussed later in this module.

## Adding Commits



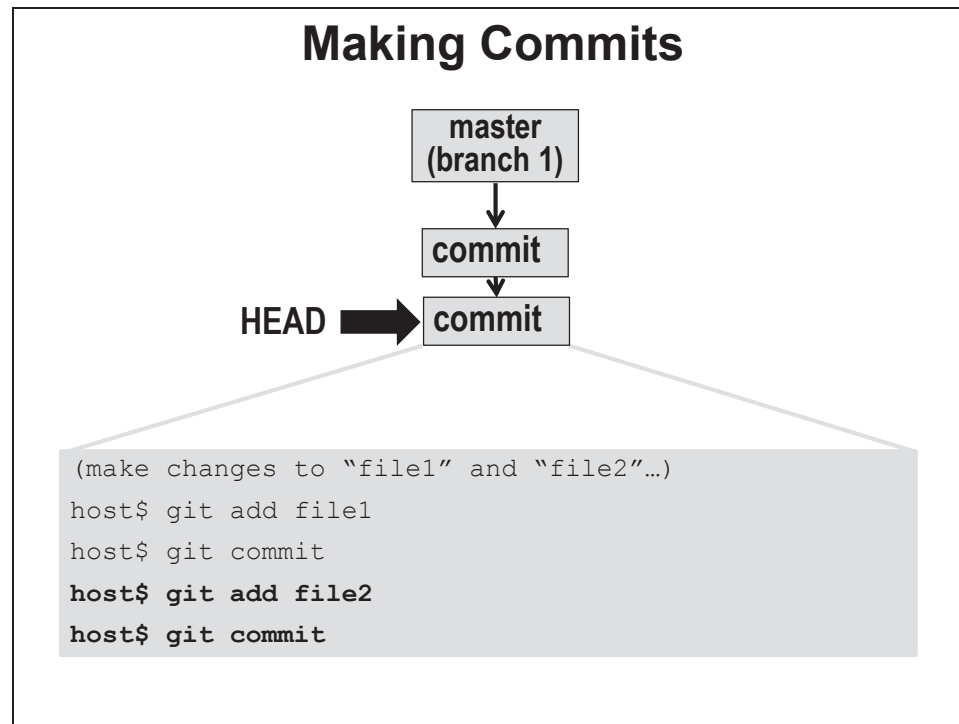
The current position within the git repository is maintained by the “HEAD” pointer. When the repository is first created, this HEAD pointer is located on the master branch, which is empty. At this point the user may create a new commit node by first using “git add” to add the files into the repository and then calling “git commit” to create the commit node. The “-m” command may optionally be used to provide a message for the commit that will appear in the log.



When a new commit node is created, the HEAD pointer automatically updates to the new commit node. In the example above, both “file1” and “file2” are modified. Another commit node is then made by calling “git add file1” followed by “git commit.” Note that, even though both “file1” and “file2” have been modified since the previous commit node, only the changes to file1 will be brought into the new commit node, because only this file was added using the “git add” command.

When “git add” is called, it places the specified files into the git repository “staging area.” Only the files in this staging area are added into a commit node when “git commit” is called, and the staging area is cleared after each “git commit” call is used to create a new commit node, so “git add” must be called before each call of “git commit” to specify the files which will be committed.

There is also a shortcut, which is to call “git commit” with the “-a” flag, which will commit the changes to all files that are in the git index. The index contains all files that have ever been added into the repository using “git add.”



The previous example is continued on this slide. Since both “file1” and “file2” were modified, but only the changes to “file1” were committed, the user could next call “git add file2” and create a second commit node that contains the changes to file2.

## Commits and SHA1 sums

**Each time a commit is made, an SHA1 hash operation is used to assign the commit a unique identifier.**

```
host$ git log
commit b87bd642de3c1d6a47cde7bfa9305a29606c7781
Author: user <user@workshop.net>
Date:   Wed Aug 1 11:49:53 2012 -0700

    initial commit
```

**A commit can be retrieved using the checkout command with its SHA1 sum**

```
host$ git checkout b87bd642de3c1d6a47cde7bfa9305a29606c7781
```

Each time a commit node is created, the SHA1 hash algorithm is used to create a unique identifier by hashing the committed files. This hash identifier appears in the log, which can be accessed using the “git log” command.

The user can revert to a previous commit node using the “git checkout” command and specifying this hash as a parameter. The entire hash does not have to be entered; if the user enters enough of the hexadecimal hash to uniquely specify the hash, this is sufficient. In the above example, the user could check out the same commit node using “git checkout b87b,” assuming that these four hexadecimal digits were sufficient to uniquely specify the hash.

## git tags

**Checking out commits by their SHA1 sum is tedious and the SHA1 sum is not particularly informative.**

**Instead, one may create a tag to reference the commit**

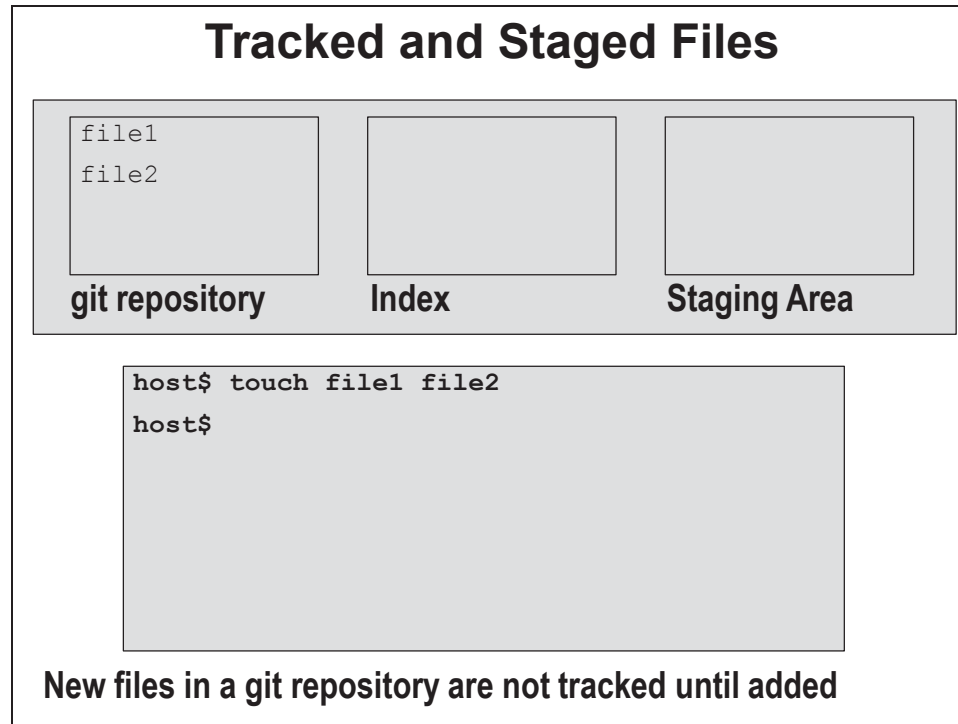
```
host$ git tag -a myTag -m "message"  
b87bd642de3c1d6a47cde7bfa9305a29606c7781
```

**This tag is now an alias for the commit's SHA1 sum**

```
host$ git checkout myTag
```

Even with the shortcut of specifying a commit node hash with only the number of digits required to disambiguate from the other commit node hashes, it can be tedious to check out nodes with their commit hashes. git allows the user to create aliases for these hash values using the “git tag” command.

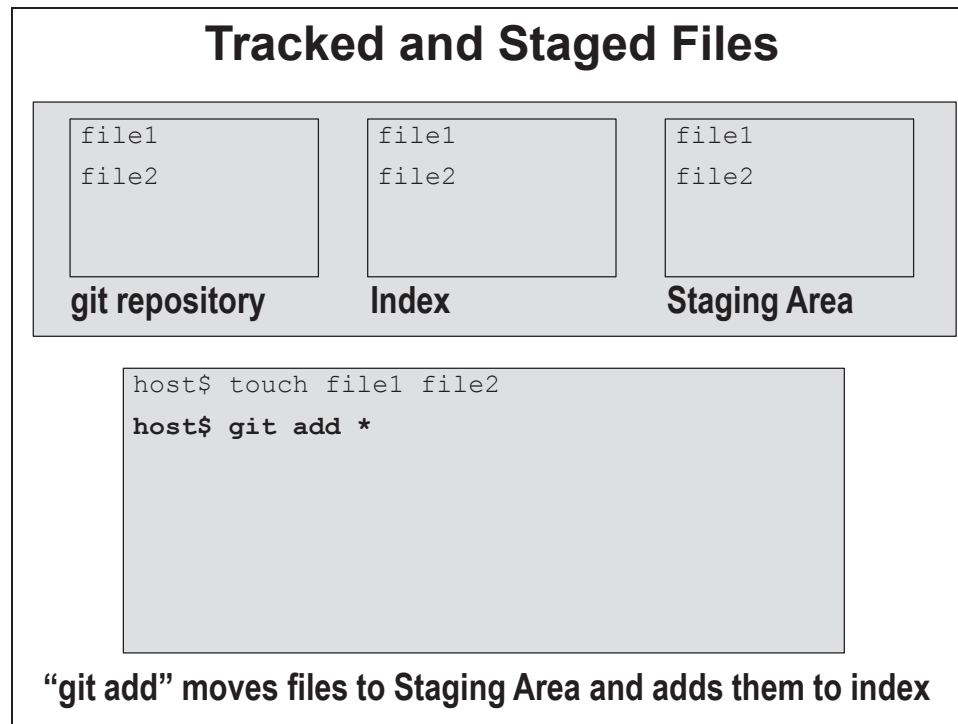
In the above example, the alias “myTag” is created for the commit node hash value, and the user can check out the commit node using this alias instead of the sha1 hash. In the above example the commit node to be tagged is manually specified by its hash. If the hash were left off, the git tool would tag whichever commit node was currently checked out, i.e. whichever commit node to which the HEAD pointer is currently pointing.



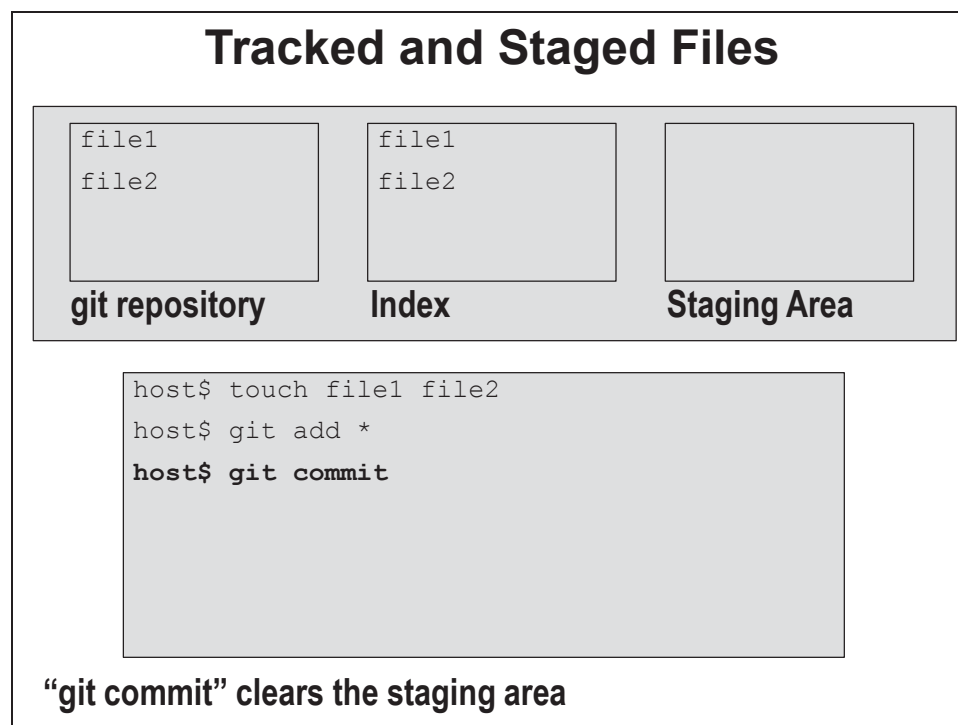
This slide provides a visual representation of three states that files in the repository may be categorized within. Here “git repository” indicates a file that is within the directory that the git repository is created in, but has not been indexed or staged. The “index” is a listing of files that are tracked by the repository, which is any file that has ever been “added” into the repository. Finally, the “staging area” indicates the set of files that are currently staged to be committed in the next call of “git commit.”

In the first step, two empty files “file1” and “file2” are created using the touch command, which is generally used to update the timestamp of a file, but will also create the file if it does not exist. These files are created in the git repository directory, but have not yet been added to the index or staging area.

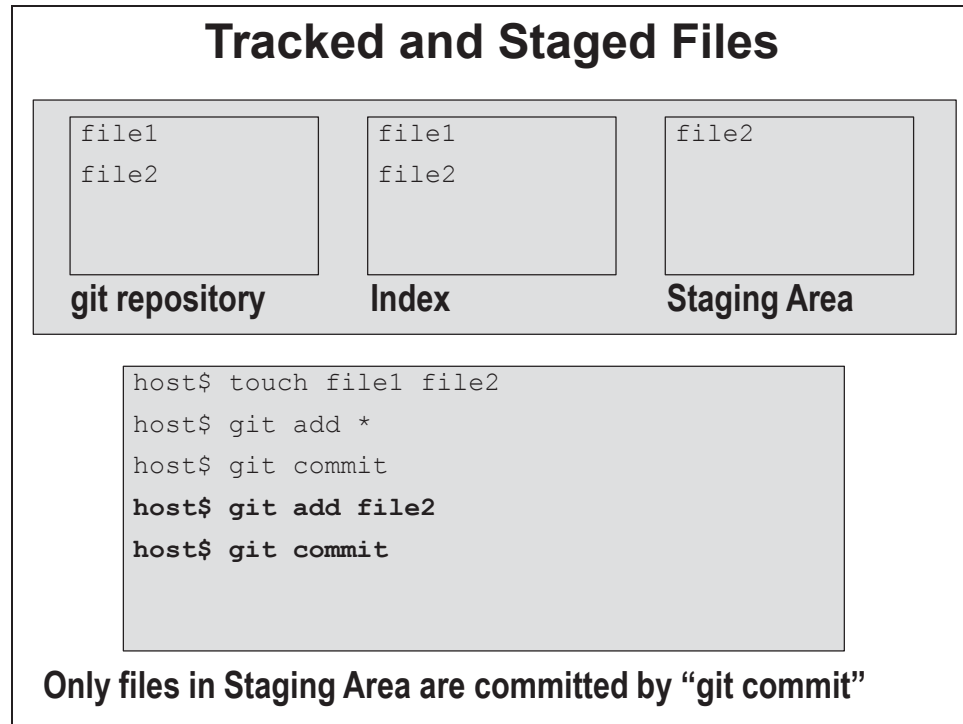




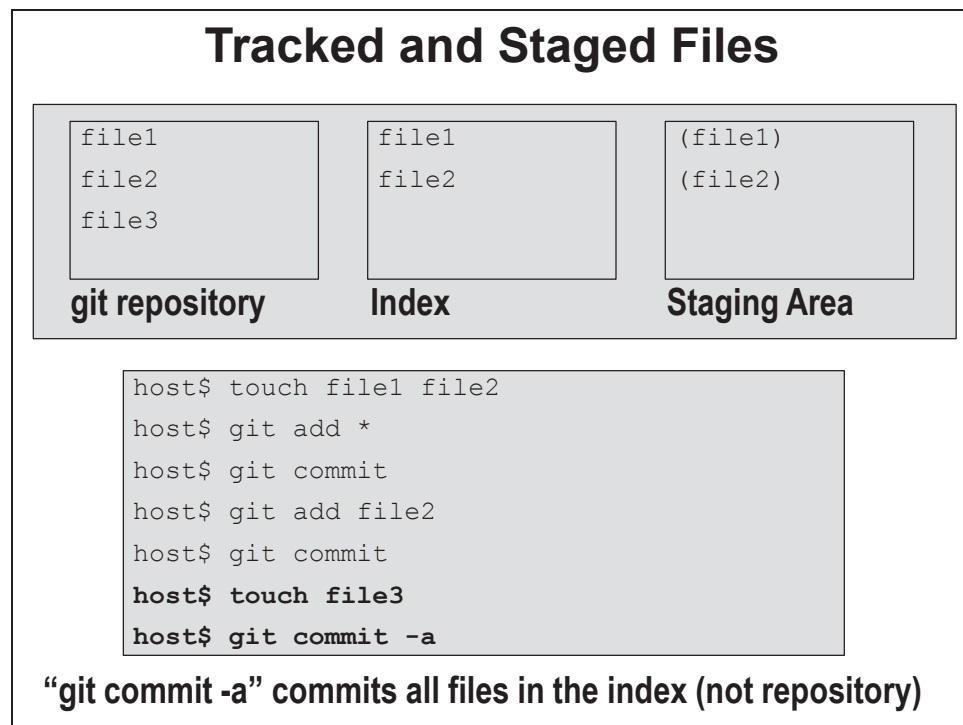
Next the “git add” command is used to add “file1” and “file2” to the git repository. This adds both files to the index as well as adding them to the staging area.



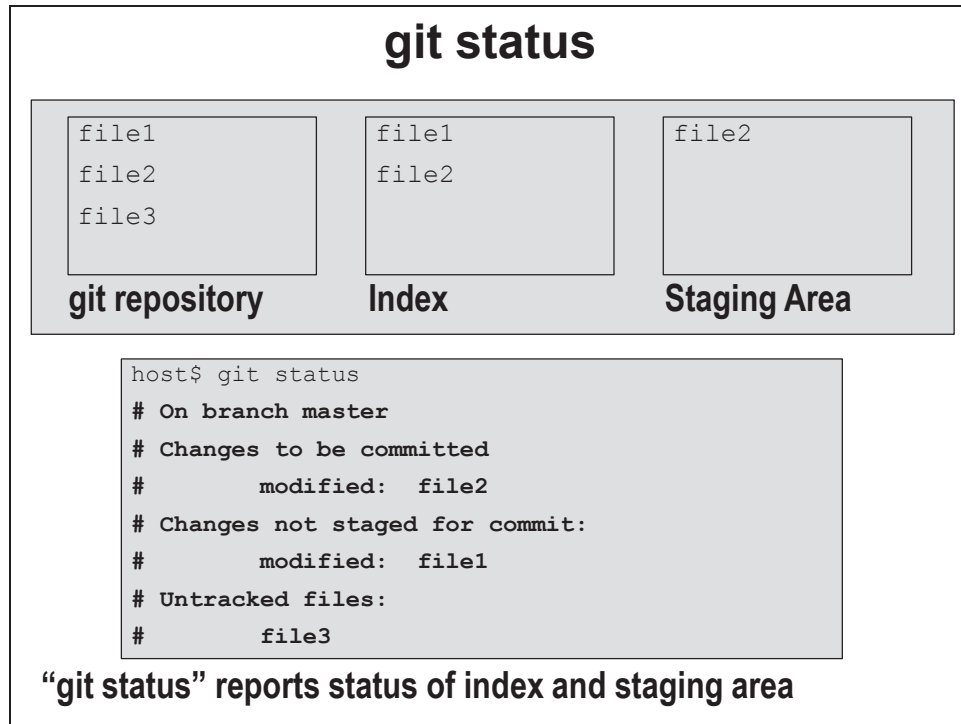
The “git commit” call commits all files that are in the staging area and subsequently clears the staging area. file1 and file2 remain in the index, however.



When “git add” is called with file2, it is added to the Staging Area (as well as to the index, although it was already in the index.)

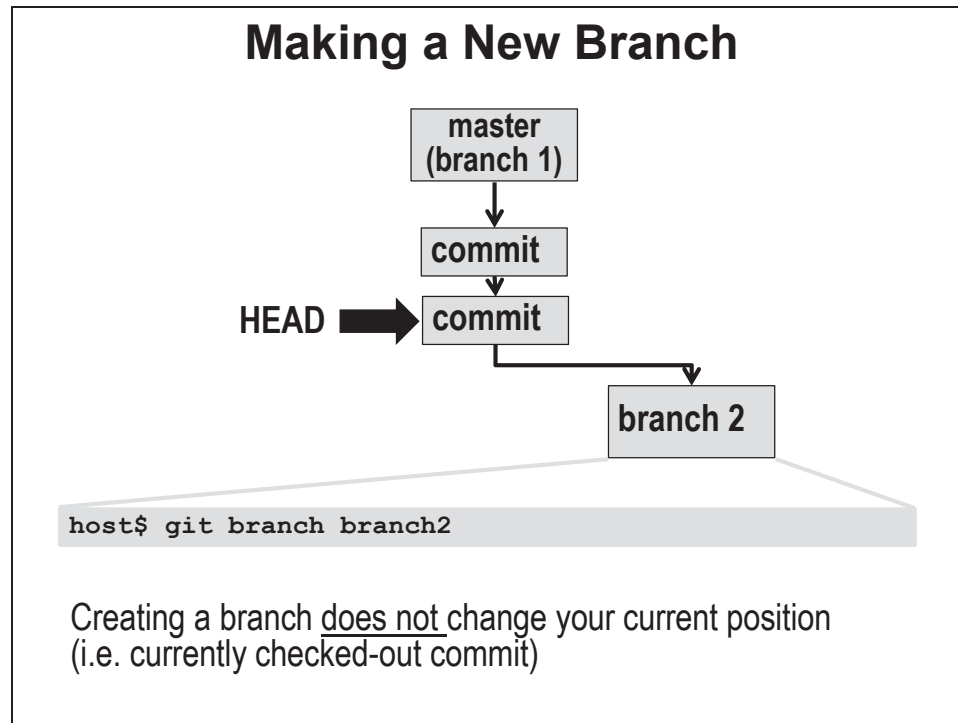


A new file is created, file3, but file3 is not added using “git add,” so it is in the git repository, but not in the index or the staging area. “git commit” is called with the “-a” option, which commits all files that are in the index, even though the staging area is empty.



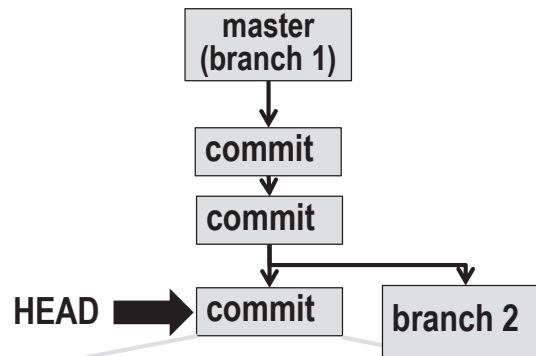
The “git status” command can be used to determine which files are in the repository, index and staging area. “Changes to be committed” reports the files that have changes and are currently in the staging area. “Changes not staged for commit” indicates files that have changes and are in the index, but are not in the staging area. “Untracked” files indicates files that are in the git repository directory but have not been added to the index.

## Adding and Merging Branches



New branches are created using the “git branch” command, followed by the name of the branch to create. The branch is created from the currently checked out commit, i.e. from the location of the HEAD pointer. Note that, while calling “git commit” automatically updates the HEAD pointer to the newly created commit, calling “git branch” does not, by default, update the HEAD pointer to the newly created branch.

## Making a New Branch

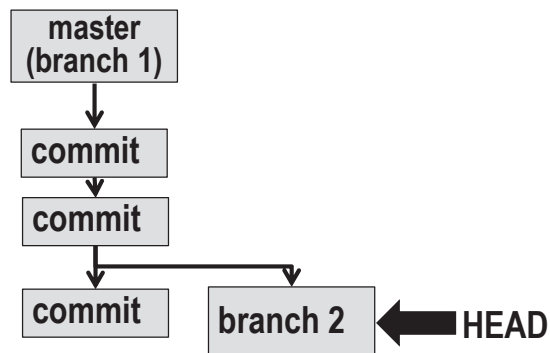


```

host$ git branch branch2
(change file1)
host$ git commit -a
  
```

Because the HEAD pointer remains on the mater branch, the next call of “git commit” creates a commit node on the master branch.

## Making a New Branch



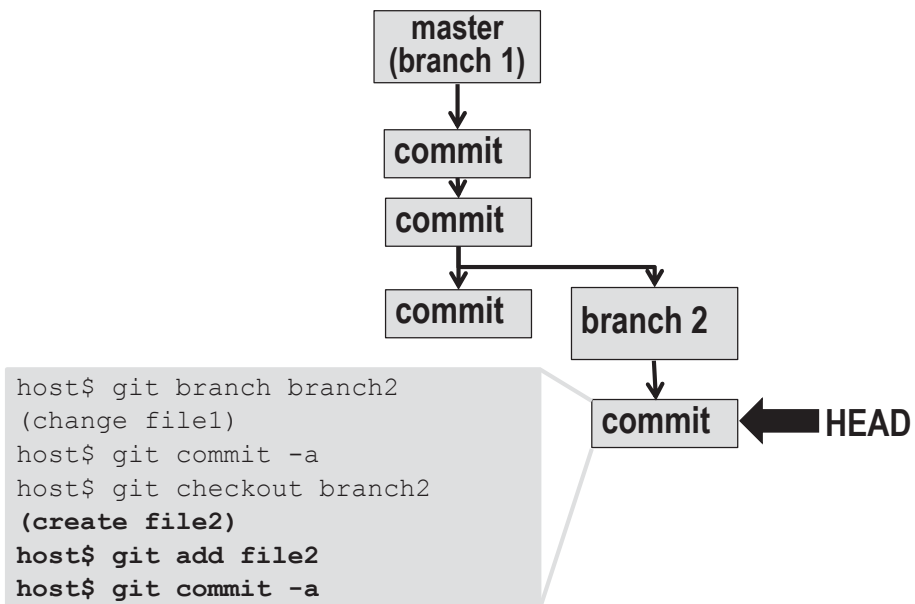
```

host$ git branch branch2
(change file1)
host$ git commit -a
host$ git checkout branch2
  
```

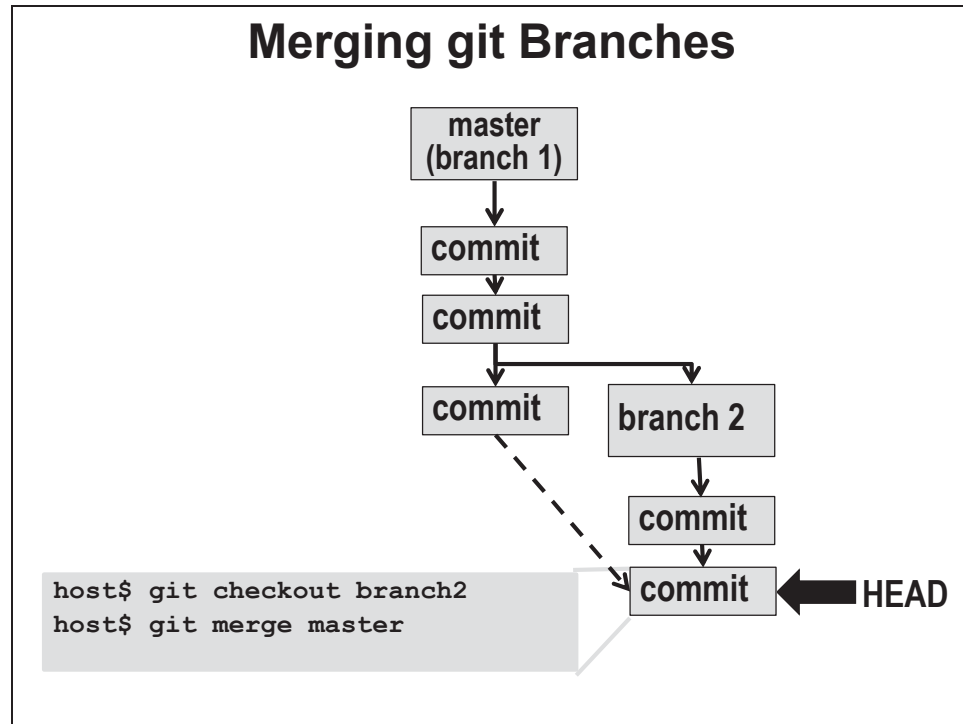
Use “git checkout” to move to new branch

“git checkout” is used to move the HEAD pointer to branch2. Note that whenever “git checkout” is used to move to a new branch, the HEAD pointer will always move to the latest commit node on that branch.

## Making a New Branch (Example 2)



Now, when “git commit” is called, the new commit node is placed onto branch2.

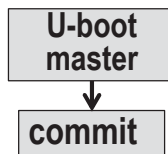


Maintaining multiple branches allows code to be developed along separate pathways, so that changes made in one branch are kept distinct from the changes made in another branch. Often, however, it is desirable at some point to combine the changes that have been made in two branches together. For instance, if multiple developers are working on the same code base by each maintaining their own branch, this allows them to work on separate tasks without interfering with each other's work. At some point, however, the contributions of each developer need to be merged together into the final code base.

This is done using the “git merge” command. The user first calls “checkout” to move the HEAD pointer to a specific branch and then calls “get merge,” specifying a second branch, which will be merged into the current branch. Note that using checkout to move the HEAD pointer to branch2 and then calling “get merge master” is not equivalent to checking out master and then calling “get merge branch2.” The difference is that only the currently checked out branch is changed by a merge operation.

If the user checks out branch2 and then calls “get merge master,” this will merge all of the new changes from the master branch into branch2, creating a new commit node that reflects all of the changes made to the code in both branches since the branch point of master and branch2. The master branch in this example, however, remains unaffected. Note that the history is preserved. A new node is created that contains the merge of the branches, but both branches remain in the git tree, as do all of the commit nodes that were created before the merge. As such, one can always revert back to a state before the merge.

## U-boot Practical Example



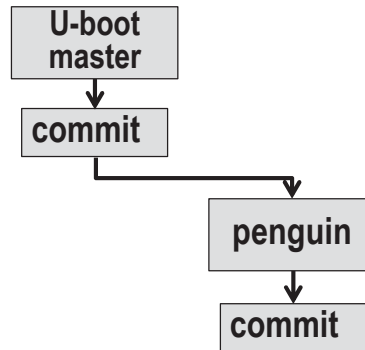
1. **Ima Pro of Penguins, Inc. clones the Arago u-boot for AM335x EVM from arago-project.org**

The module will conclude with a practical example to demonstrate why developers should clone TI's git repositories containing TI-provided code instead of simply copying a snapshot of code from the repository.

In this example, Ima Pro begins by following the recommended procedure of cloning the u-boot repository from the arago-project.org site.



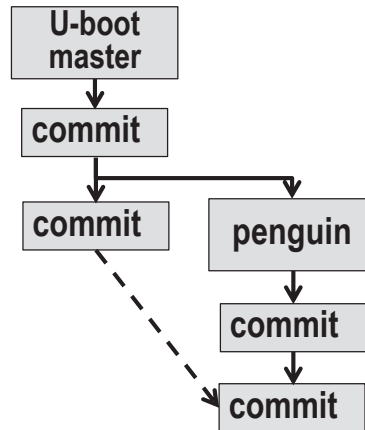
## U-boot Practical Example



1. Ima Pro of Penguins, Inc. clones the Arago u-boot for AM335x EVM from arago-project.org
2. Ima ports U-boot to the Penguins, Inc. production board

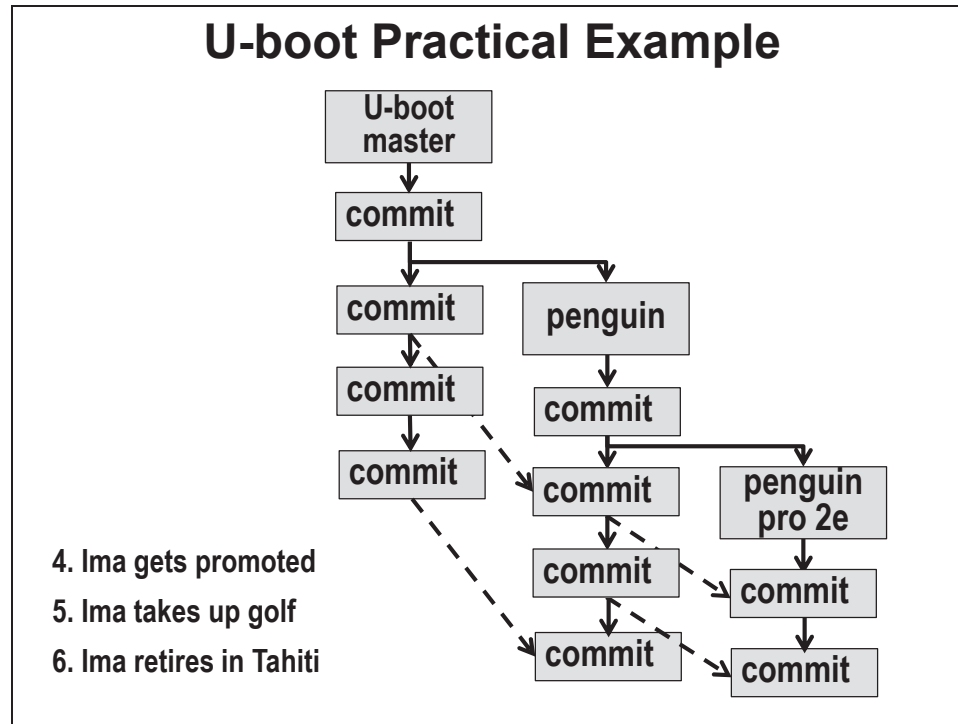
Ima then creates a new branch named “penguin” in which she ports u-boot to a custom hardware board that her company has created.

## U-boot Practical Example



1. Ima Pro of Penguins, Inc. clones the Arago u-boot for AM335x EVM from arago-project.org
2. Ima ports U-boot to the Penguins, Inc. production board
3. As U-boot is updated on Arago git, Ima merges patches

At a later time, Ima discovers that u-boot is not working properly in her final test suit preceding her release to market. After some debugging, she localizes the issue to u-boot. Ima sends a message to “support@ti.com” reporting the issue she is having with the code and is notified that the problem she is having is a known issue and that the code has been updated since she originally cloned the u-boot git tree.



Because Ima’s penguin branch is a clone of the arago-project.org git repository, she simply uses a “pull” operation to update her local repository with the latest changes to the code, which appear as new commit nodes on the master branch of her repository. She then uses the “git merge” command to merge these changes into her penguin branch. The merged commit node then contains the changes that Ima made to the original u-boot code base in order to port to her custom board as well as the changes that TI has made since she originally cloned the repository, which include the code patch that addresses the issue she found.

At a later date, Ima might even create a new branch, “penguin pro 2e,” to address a new board that is a derivative of the original “penguin” board. It would make sense to create the new branch off of one of the commit nodes of the “penguin” branch. Ima can then keep up-to-date with any new changes made by TI to the u-boot code base by pulling from arago-project.org and then merging the latest commits into her local branches.

## Lab 11: Using git

### Lab 11: Exploring git

- A. Create a new git repository
- B. Examine a workshop git repository

- A. Use “git init” to create a new repository and then “git commit” and “git tag” to save files in various states and explore the effects of “git checkout” and “git merge.”
- B. Explore the “meta-workshop” git repository from the workshop install files.